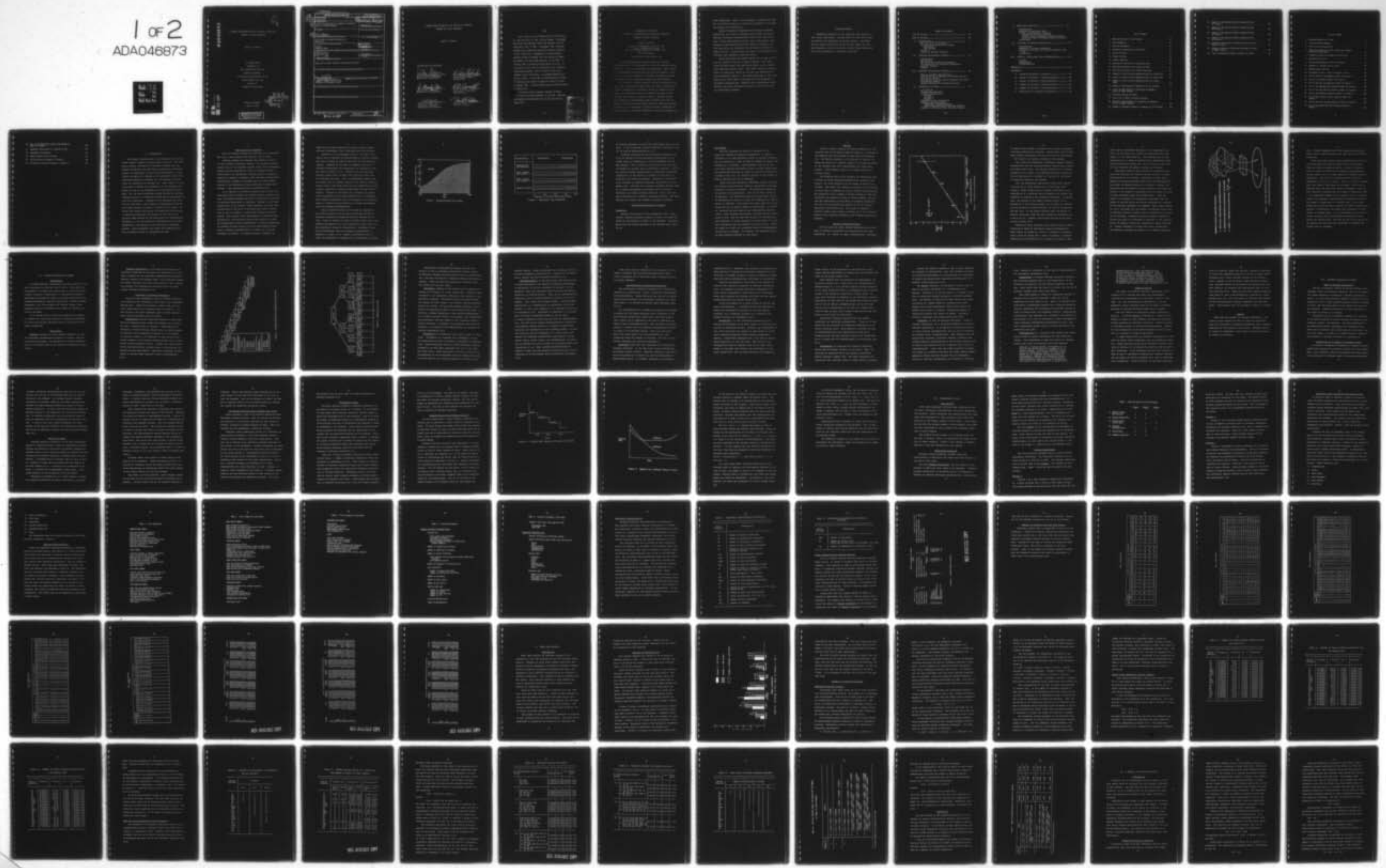
AD-A046 873   AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO   F/G 9/2
A MODEL FOR ESTIMATING THE NUMBER OF RESIDUAL ERRORS IN COBOL P--ETC(U)
JUN 77 C E MARTIN
UNCLASSIFIED   AFIT-CI-77-97   NL

1 OF 2
ADA046873



①

A MODEL FOR ESTIMATING THE NUMBER OF RESIDUAL ERRORS IN COBOL PROGRAMS

Cecil E. Martin

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fullfillment of the

Requirements for the

Degree of

Doctor of Philcsophy

Auburn, Alabama

June 7, 1977

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
(14) 1. REPORT NUMBER AFIT-CI-77-97 ✓	2. GOVT ACCESSION NO.
	~~3. RECIPIENT'S CATALOG NUMBER~~ (9) Doctoral thesis,
4. TITLE (and Subtitle) (6) A Model for Estimating the Number of Residual Errors in COBOL Programs,	~~5. TYPE OF REPORT & PERIOD COVERED~~ Dissertation
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Cecil E. Martin	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT Student at Auburn University, ✓ Auburn AL	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/CI WPAFB OH 45433	~~12. REPORT DATE~~ (11) June 1977
	~~13. NUMBER OF~~ PAGES 139 Pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 152p.	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE AFR 190-17.

[illegible]

JERRAL F. GUESS, Captain, USAF
Director of Information, AFIT

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

012200 JB

A MODEL FOR ESTIMATING THE NUMBER OF RESIDUAL ERRORS IN COBOL PROGRAMS

Cecil E. Martin

Certificate of Approval:

[illegible]	H. Troy Nagle Jr.
J. D. Irwin, Professor Electrical Engineering	H. T. Nagle, Jr., Chairman Professor, Electrical Engineering
J. Grady Cox	B.D. Carroll
J. G. Cox, Professor Industrial Engineering	B. D. Carroll, Associate Professor, Electrical Engineering
[illegible]	Paul F Parks
Dr. K. B. Cook, Jr. Assistant Professor Electrical Engineering	Paul F. Parks, Dean Graduate School

VITA

Cecil E Martin, son of Stephen and Nell (Folks) Martin, Sr., was born April 13, 1941, in Kite, Georgia. He attended Bartow and Kite Elementary Schools and graduated from Kite High School, Kite, in 1959. In September 1959, he entered Brewton Parker College and received the degree of Associate in Arts in June 1961. Subsequently, he entered Georgia Southern College in September 1961 and received the degree of Bachelor of Science (Math Education) in June 1963. In February 1964, he entered the United States Air Force. He is currently a Major and his professional title is Computer System Design Engineer. While in the Air Force, he has attended several Universities. He attended Midwestern University in 1965. In June 1967, he entered Georgia Institute of Technology and received the degree of Master of Science in December 1968. He began graduate study at Auburn University in January 1975.

He married Virginia (Ginger), daughter of Thomas Charles and Rae (Fried) Grabfelder in July 1965. They have two daughters, Diane Michelle and Lisa Ann, and one son, Thomas Cecil.

DISSERTATION ABSTRACT

A MODEL FOR ESTIMATING THE NUMBER OF RESIDUAL

ERRORS IN COBOL PROGRAMS

Cecil E Martin

Doctor of Philosophy, June 7, 1977
(M.S., Georgia Tech, 1968)
(B. S., Georgia Southern College, 1963)

151 Typed Pages

Directed by H. Troy Nagle, Jr.

The most significant problem facing the computer profession today is manifested in two major complaints about software: it is too expensive and unreliable. Most computer professionals recognize the high cost as largely a symptom of the latter complaint. The high incidence of errors in software is the underlying reason for unreliability.

The number of errors uncovered during the software life cycle has a significant impact upon the cost in terms of resources (personnel and computer) needed to correct the errors. The correction cost is a function of when, in the software life cycle, an error is found. Software errors found during the development phase generally cost less to correct than errors which occur during the operations phase. Therefore, it is necessary to detect software (program) errors as early as possible--preferably before the software is made operational. Also, it is necessary to predict the number of residual errors in a program to determine if and when the program goes operational.

Program structural characteristics metrics (internal complexity) is a means of estimating the number of errors. Thirteen unrelated characteristics metrics are used to define 7 local complexities--control flow, input/output, data handling, computational structural design, interface, and data use--which are predictors of the number of errors in COBOL programs. Linear models for each of these metrics are available for predicting software errors.

Models developed from these metrics can be used to predict the number of errors in COBOL programs. The "best" single variable model for predicting errors is the Control Flow Complexity metric model. The "best" multiple variable model for predicting errors is the one that contains all 7 local complexity metrics. The latter model can be used when dealing with many types of programs that are developed by different organizations. However, each organization should estimate the model parameters relative to error data from its development projects.

ACKNOWLEDGEMENTS

Performing research of this magnitude and nature is a very large undertaking. Success would not have been possible without the assistance of several people. Therefore, special thanks is given to Donald Wright, Robert Ben and Bill Eiseman of the Air Force Data System Design Center for all their assistance in obtaining error data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The primary consideration in any system is that it performs properly whenever the user wants to use it. For computer systems, consisting of hardware, software and man-machine interfaces, the most widely accepted and meaningful measure of performance is total system reliability. Total system reliability is defined as the probability that every subsystem performs as intended for the necessary time and under the conditions of customer use. Thus, there is an obvious need to measure the reliability of the software subsystem as well as the hardware and the man-machine subsystems. But the most significant problem facing the computer profession today is a software problem that is manifested in two major complaints: software is too expensive and software is unreliable. Most software professionals recognize the former problem as largely a symptom of the latter. Although this paper's main focus is primarily on the problem of unreliable software, the problem of high cost is an indirect issue because of its relationship to unreliability. Therefore, to put the problem of unreliability into proper perspective, the problem of high cost will be discussed briefly. This discussion will stress the important role that reliability plays in increasing the cost.

The High Cost of Software

There are several reasons why software is so expensive. The rest of this section will discuss a few of them.

Computer systems are becoming more complex as faster and more versatile hardware evolves. The resultant sophisticated uses of the computer systems demand that programmers develop reliable software to drive the computer system. Additionally, the man-machine interface which is generally handled by software is becoming more and more sophisticated. Consequently, software is becoming more and more complex because of the hardware and the man-machine interface subsystems. This increases software development cost.

As we continue to automate processes which control our life-style--bank accounts, air traffic control, medical systems, and defense systems--we have to trust more and more in the reliable functioning of software. Nowhere is this more evident than in the military where computers are being used increasingly as the heart of sophisticated weapon systems such as the B-1 bomber or a real-time command and control system. They control their environments by receiving data, processing it and returning results fast enough to affect the functioning of their environments. Reliable functioning of software is also critical in an on-line banking system where a software error(failure) may result in a loss of thousands of dollars. To develop reliable software, we spend more and more resources on quality control during software development, thus, increasing the cost directly.

Software is a big business in the U. S. today. The annual cost of software is approximately 20 billion dollars. Its rate of growth is greater than that of the economy in general. Compared to the cost of hardware, the cost of software--development and maintenance--is escalating along the lines in Figure 1 [1]. Studies [2,3] indicate that software demand over the years 1975-1985 will grow about 21-23 percent per year. This is considerably faster than the growth rate in software supply at the current estimated growth rates of the labor force and its productivity per individual which has a combined growth rate of about 11.5-17 percent. Because of the demand and a shortage of experienced programmers, error prone software will be developed. Poor software reliability will be revealed by an excessive number of software errors resulting in higher maintenance cost and customer dissatisfaction.

Errors discovered after the software is operational will impact greatly upon the cost of software because of computer resources and manpower needed to correct the errors. About 70 percent of today's software dollar goes into software maintenance, and this number will likely grow [4]. This percentage varies by organization. Maintenance cost versus development cost for different organizations is depicted in Figure 2 [5]. DeRose [6] estimates that it costs the Department of Defense $75 an instruction to devel-

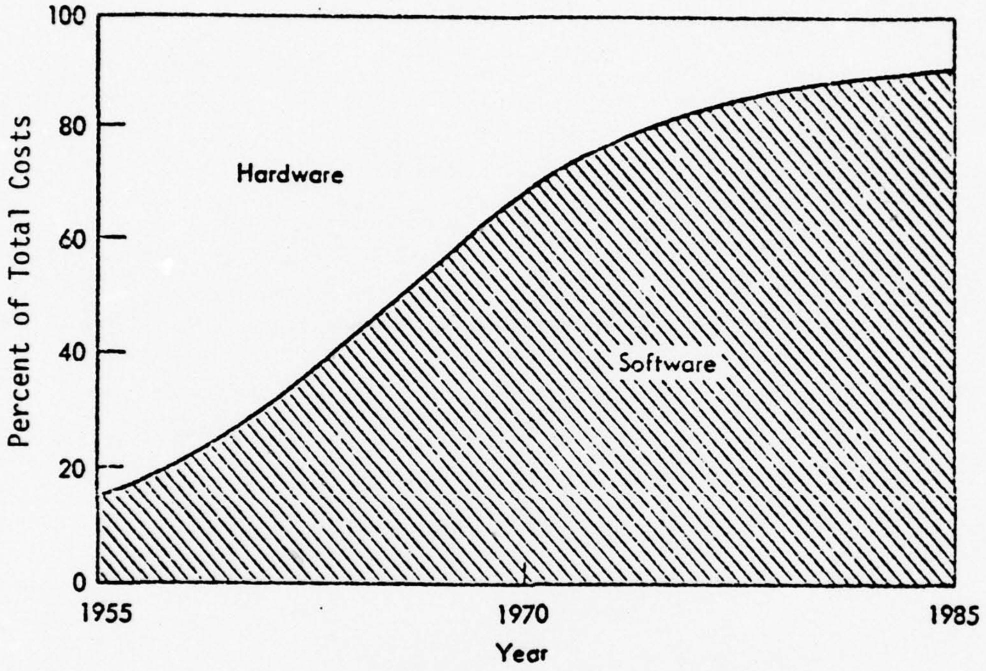


Figure 1. Hardware/Software Cost Trends.

```
r-----------------+---------+---------+---------+---------+,
|                 |                                        |
|Organization     |    Maintenance           Development   |
|                 |                                        |
|-----------------|                                        |
|                 |                                        |
| General Tel.    |----------------------------------------|
| and Electric    |........................////////////////|
|                 |----------------------------------------|
|                 |                                        |
| USAF Command    |----------------------------------------|
| and Control 1   |.........................../////////////|
|                 |----------------------------------------|
|                 |                                        |
| USAF Command    |----------------------------------------|
| and Control 2   |.............................///////////|
|                 |----------------------------------------|
|                 |                                        |
|                 |----------------------------------------|
| General Motors  |..............................//////////|
|                 |----------------------------------------|
|                 |                                        |
L-----------------+---------+---------+---------+---------+J
                  0        25        50        75       100
                   Percent of 10-Year Life-cycle Costs
```

Figure 2 Life-cycle Cost Breakdown.

op aviation software, but that the maintenance cost is a lot more. On one particular aircraft computer, maintenance cost ran as high as $4000/instruction [7].

Proposed solutions to the cost problem invariably involve an attempt to raise programmer productivity by devising tools and techniques to allow programmers to work more quickly. But it should be obvious that the high cost of software is largely due to reliability problems. Cost is not usually lowered significantly by increasing programmer productivity if the latter is a measure of the speed of designing and coding the program. Depending on the situation, attempts to increase programmer productivity can increase cost. The best way to sharply decrease software cost is to reduce maintenance and testing cost by devising techniques to produce reliable software. This is the primary motivation for a software reliability theory. The next section will discuss the software reliability problem.

The Software Reliability Problem

Definition

Software reliability is the probability that a given program operates correctly, without an error, for some time period on the machine for which it was designed. Correctly means that the program performs as the ultimate user wants it to.

The Problem

The high incidence of errors in software is the underlying problem of software reliability. It would be fortunate if the well-developed theory of hardware reliability (see Appendix A) could be used to predict or enhance the reliability of software. Unfortunately, this is not the case since hardware reliability theory is based mainly upon the statistical analysis of random and wear-out failures of components with age. In contrast software is not subject to wearout failures once it is debugged.

There are other important differences between hardware and software which make the hardware reliability techniques difficult to apply to software. The elementary components of software are instructions. They do not wear, break, or deteriorate. All software errors are in some sense design or implementation errors [3] which are comparable to burn-in errors in hardware. When errors are found, they can be corrected and are no longer present in the program. In general, programs are more complex than corresponding hardware logic. Large programs are probably the most complex objects built by man. Some of them have millions of instructions. The complexity of these programs is so great that it is not well understood what the program can or can not do. Finally, there is a lack of a scientific basis for understanding the nature of programs. In contrast, the scientific basis of most hardware elements is well known.

Purpose

Figure 3 shows a summary of current experience on the relative cost of correcting software errors as a function of the software life cycle phase in which they are corrected [5]. For obvious reasons, it is desirable to predict the number of errors in a software system at the earliest moment in the software life cycle (development and operational phases). Unfortunately there is no proven technique in practice today.

The research done to date suggests the hypothesis that profiles of actual program characteristics (internal complexity) are good predictors of the number of errors in a program. This paper will present the results of an analysis of error data to determine if actual program characteristics are predictors of the number of errors, propose a model for predicting the number of errors in COBOL programs, and discuss the application of this model to software reliability. The rest of this Chapter and Chapter II and III contain background information only. The reader is directed to continue reading this report at Chapter IV if he is already familiar with software engineering and software reliability concepts.

Survey of Related Research

Over the past ten years, several investigations in the area of software reliability and phenomenology have been undertaken. As a result of these investigations, reliabili-

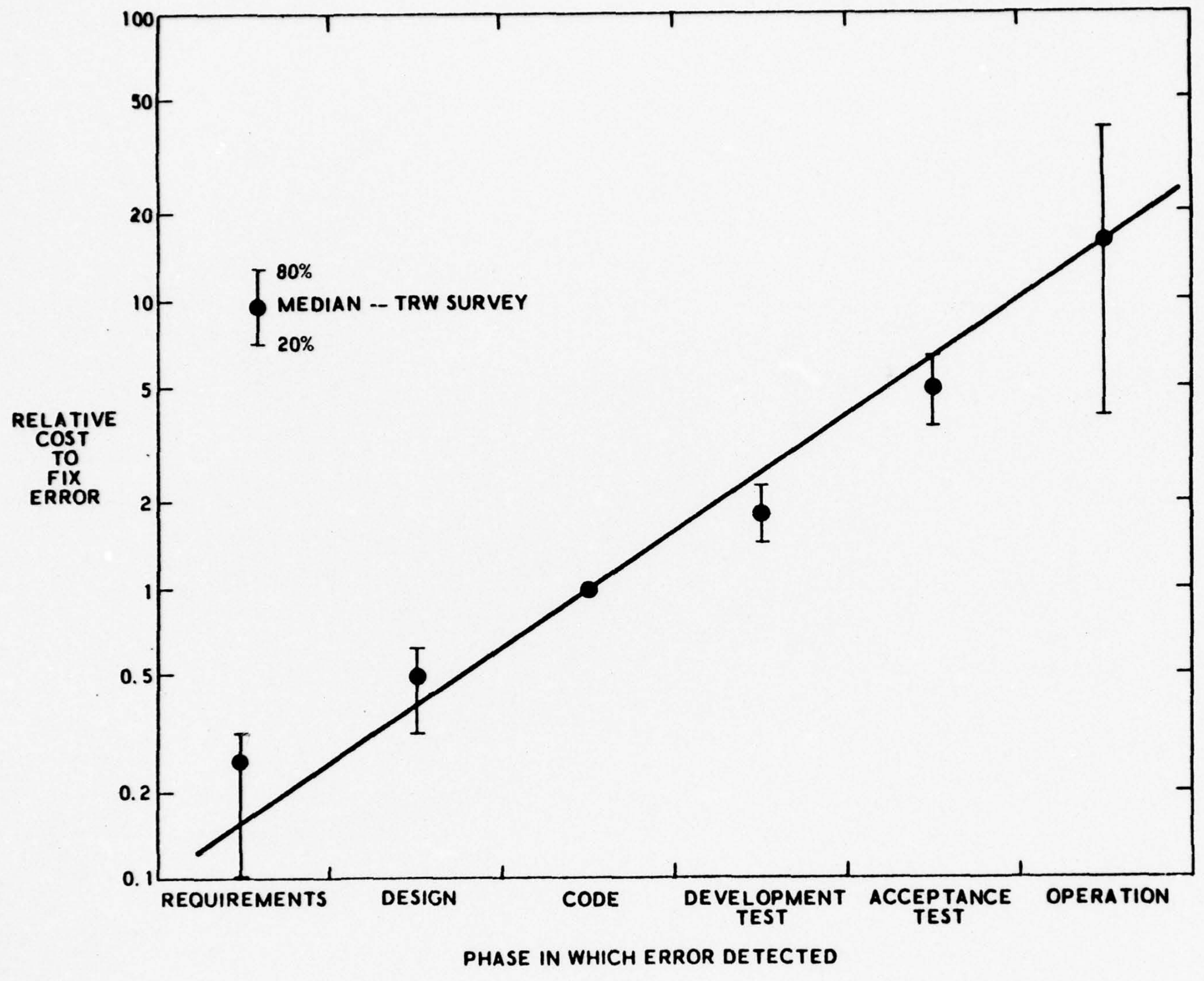


Figure 3. The Price of Procrastination

ty models which attempt to describe the failure of software have been proposed and discussed. These models were derived from hardware reliability and have not been very successful . This failure is primarily founded on two reasons.

One, there are fundamental differences between software phenomenology and the hardware-oriented assumptions on which the models were based. The failure mechanism of a hardware component is by chance or by component wear-out whereas the failure mechanism of a program is a function of the number of remaining errors in the program.

Two, the fundamental statistical issues which emanate from the use of these models have, by and large, been ignored. These issues pertain to model verification, the development of a procedure which formalizes the testing and debugging of software, and parameter estimation. In particular, the success of the models depends largely on the estimation of the original number (N) of errors in software and the constant of proportionality (K) used in determining failure rate. Of the several methods used, the method of maximum likelihood gives the most reasonable estimators for N and K [8-10]. However, this method does not yield satisfactory results [10].

Models are being developed which explain previous error histories in terms of appropriate program phenomenology. These models are based on a view of a program as a mapping from a space of inputs into a space of outputs; of program operation as the processing of a sequence of points in the input space, distributed according to an operational profile; and of testing as a sample of points from the input space, [11,12] (see Figure 4). This approach can be used conceptually as a means of appropriately conditioning time-driven reliability models [5]. But, we still are not able to truly estimate the number of errors in software.

Additional insights into reliability estimation have come from analyzing the software errors relative to actual characteristics of programs. Currently, it seems that a measure of program complexity offers the best estimator for the number of residual errors in a program. Akiyama [13] concludes that the number of program errors is strongly correlated to the number of conditions plus the number of calls to other programs rather than program size. Lipow and Thayer [14] suggests the interesting hypothesis that the number of program errors can be best predicted by a measure of the internal complexity of programs. They, using empirical data, concluded that the number of software errors found in programs written in JOVIAL could be predicted by the number of branches, a measure of program internal complexity. Herndon and Lane [15] developed an approach to the quantification of software errors as a function of module complexity. Module complexity is based upon module composition. The complexity measure was shown to be a useful managerial

MINIMUM-VARIANCE UNBIASED ESTIMATOR

- **PICK N (SAY, 1000) RANDOM, REPRESENTATIVE INPUTS**
- **PROCESS THE 1000 INPUTS, OBTAIN M (SAY, 3) FAILURES**
- **THEN R = PROB (NO FAILURE NEXT RUN) =** $\frac{N - M}{N} = 0.997$

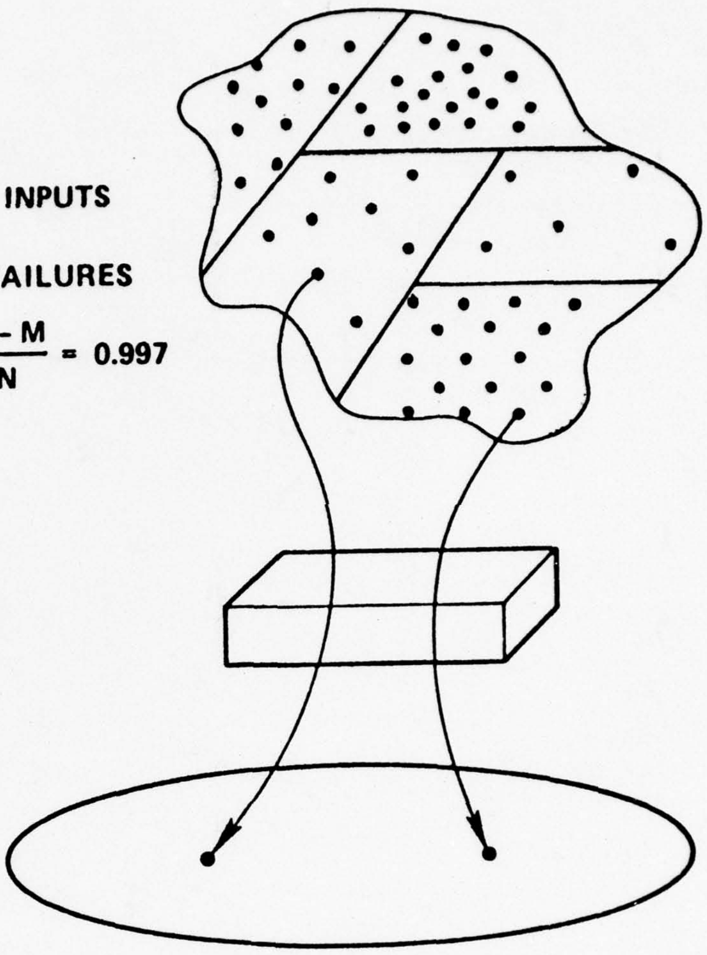


Figure 4. Input Space Sampling Provides a Basis for Software Reliability Measurement

tool. Program components with high complexity indicators should receive more attention than ones with low complexity indicators.

There have been several other investigations into program complexity that did not address the error problem. These are briefly summarized below. Flynn [16] suggests that the number of nodes in the smallest path-isomorphic program scheme may be a useful measure of inherent program complexity. Sullivan [17] proposes several complexity measures--c1, c2, c3, p1 and p2. The c1, c2 and c3 measures deal with control flow graphs of programs. The p1 and p2 measures deal with data flow graphs of programs. This report basically concludes that the number of conditions plus 1 is a complexity measure of the control flow of a program. McCabe [18] develops a graph-theoretic complexity measure--the number of conditions in a program plus 1. He illustrates how it can be used to manage and control program complexity. Additionally, he proves that complexity is independent of program size. It is appropriate at this point to stress that most all of the software reliability models employ the program size. This may be one of the reasons why the models have not been very successful in modeling the failure rate of programs.

II. SOFTWARE ENGINEERING CONCEPTS

Introduction

The TERM "software engineering" was made popular by two NATO conferences in 1968 and 1969 [23,24]. Since then the development of software has evolved into an engineering discipline involving a multiplicity of specialized branches--Requirements Engineering, Theory of Program Structures, Programming Methodology, Software Reliability, Software Project Management, etc. This chapter will briefly discuss those concepts relative to estimating the number of residual errors in programs.

It is perhaps best to view this chapter as an attempt to identify the underlying concepts of software engineering in a form that permits the main issues of this paper to be better understood.

Definitions

Software includes not only computer programs, but also the associated documentation required to develop, operate, and maintain programs. The generation of timely documentation is an integral part of the software development process [5,25].

Software Engineering is the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them. This definition covers the entire software life cycle (see Figure 5), thus including redesign and modification activities which are often called "software maintenance" [5].

The Goals of Software Engineering

There are four fundamental goals of software engineering: modifiability, efficiency, reliability, and understandability [26]. Boehm [27] provides a larger list which he calls characteristics of software quality (Figure 6). In what follows, this paper addresses some of these important goals, those considered basic in nature.

Modifiability implies controlled changes in which some parts are unchanged while others are altered, all in such a way that a desired result is obtained. Modifiability is difficult to achieve because changes occur for many reasons. For example, when transferring software to a new computer or operating system, it is desirable to keep invariant the logical effects of the system, limiting changes only to necessary machine-dependent aspects. Changes are also required to add new capabilities, correct errors in the program, and improve software performance. Different approaches are necessary to satisfy these different types of modifiability [26].

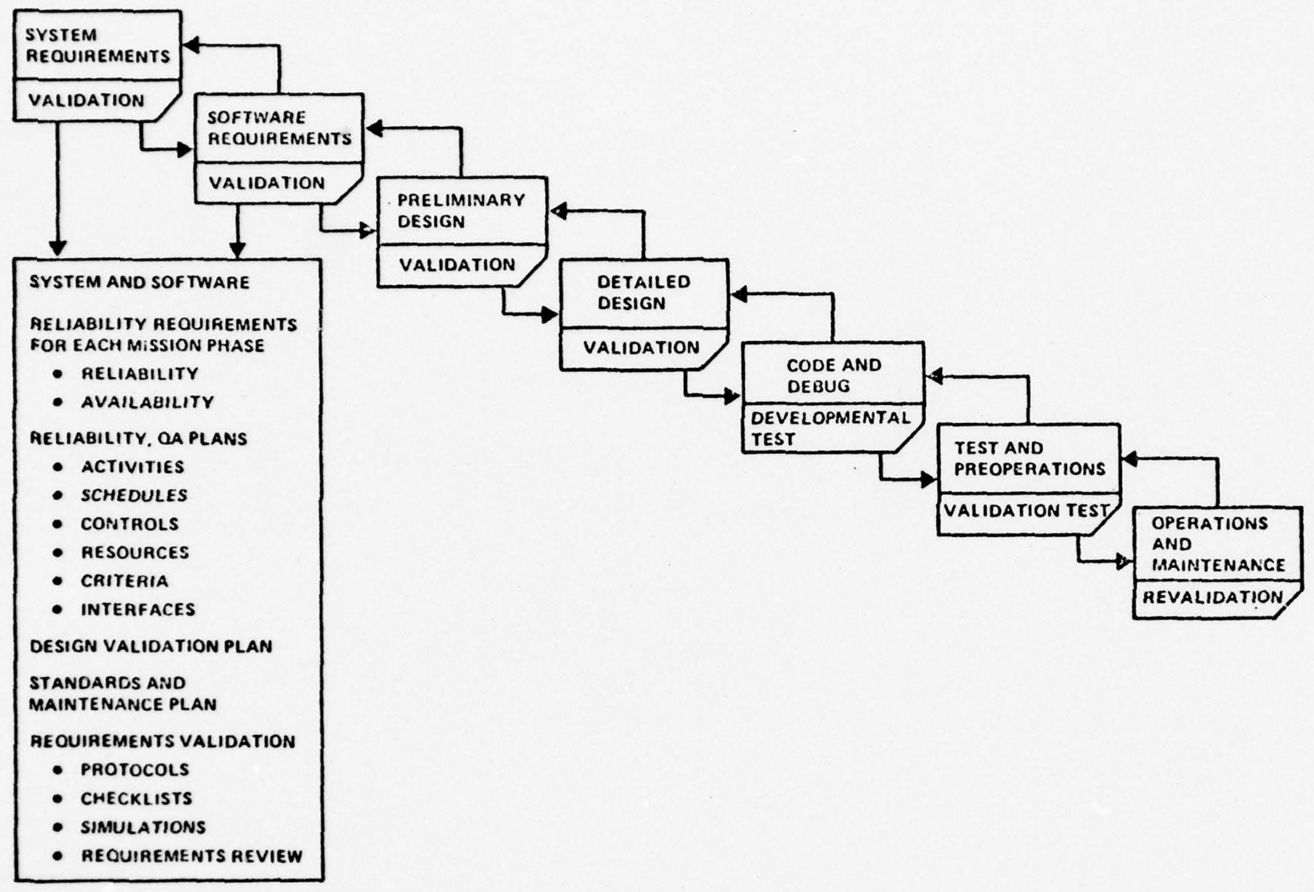


Figure 5. Software Reliability Oriented Life Cycle Plan

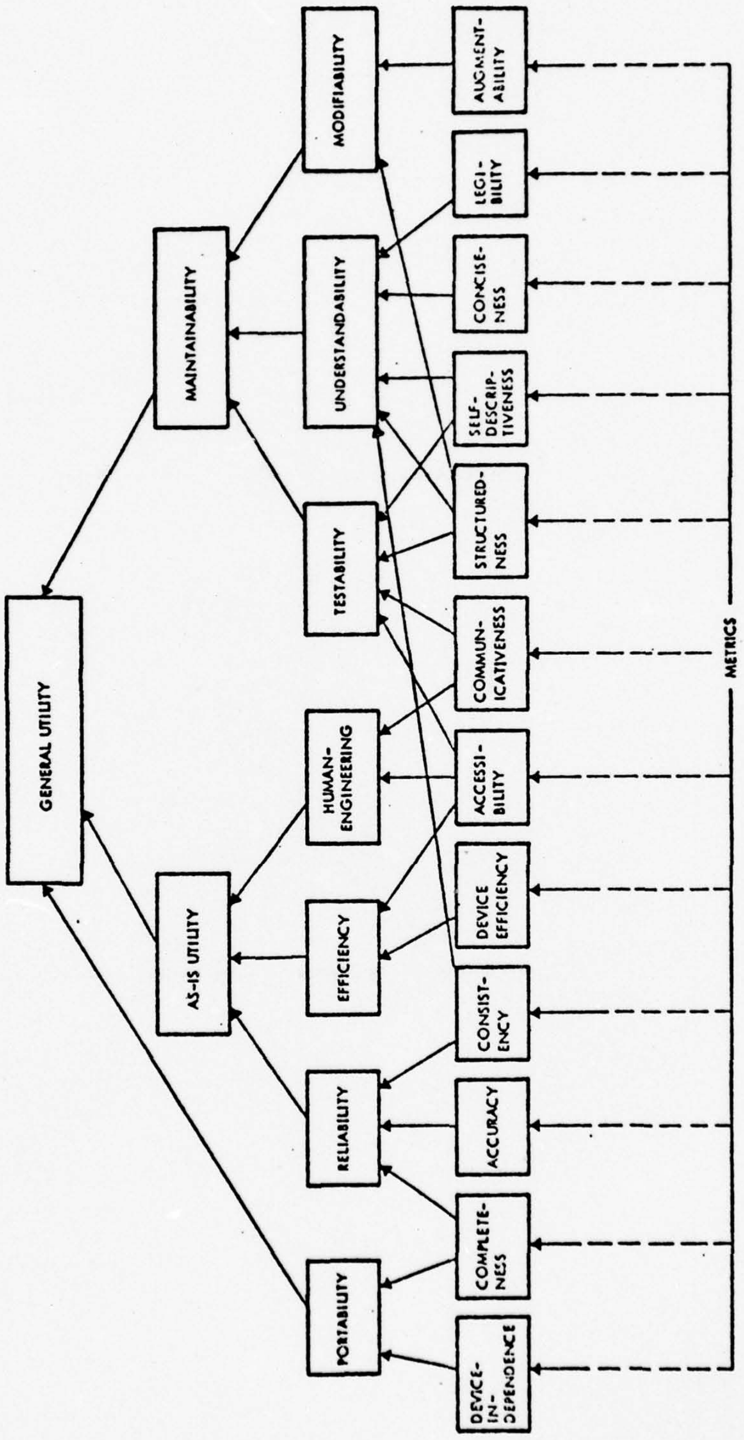


Figure 6. Characteristics Tree.

Additionally, modifiability implies not only the ability to have an adaptable evolutionary design, employ standardized software building-blocks, tune for performance, etc., but also the ability to maintain project schedules and budgets. There has been much progress in achieving this goal within the past ten years.

Efficiency, defined as the optimal use of computer resources by a program, is a much abused goal. Primarily this is because it is prematurely assigned a high priority in engineering tradeoffs. Efficiency should be treated within the context of other issues. For example, achieving modifiability can provide the basis for meeting efficiency goals during the maintenance phase of the software life cycle. In addition, insights reflecting a more unified understanding of a problem have more impact on efficiency (via abstraction and uniformity) than any amount of "bit twiddling" within a faulty structure. In general, the efficiency goal does not dominate, as reliability and modifiability, the practice of software engineering [26].

Reliability is an important goal which is much in vogue today. Reliability is concerned with conception, design, and construction as well as failure in operation or performance. Unlike efficiency which is often prematurely applied, reliability is more often considered too late in the software life cycle. Since reliability can only be built in at the beginning of the development cycle--it cannot be an add-on at the end--it is a primary problem to be solved in any software system. Hence, reliability has a crucial effect on software engineering practices [26]. Because of its importance, Chapter III will be devoted entirely to it.

Understandability is the final basic goal which exerts a strong influence in all aspects of software engineering. In particular, it is not a property of legality. It is, therefore, much more important since the entire conceptual structure is involved [26,27]. Also, in any circumstance an acceptable level of understandability either is or is not present. Thus, there is no middle ground. Although understandability is a prerequisite to reliability and modifiability, it also draws attention to an important barrier to it--complexity [26]. Management of complexity is a crucial part of software engineering methods, and the need to manage complexity arises from the goal of understandability. The only way to achieve understandability relative to an inherently complex system is to impose an appropriate structure and organization on the software system. As such, the structure must be represented in a clear notion that permits the different translations (requirements, design, source coding, object coding, and documentation) to bridge the gap between the actual system and an understandable representation of it. Thus, achieving understandability depends as much upon the software engineering tools such as compilers as on the methods such as structured programming [26].

Other goals such as portability and testability are of lesser importance than the ones discussed above. Boehm [27,28] discusses all of the above goals as characteristics of quality software.

The Principles of Software Engineering

The principles of software engineering are modularity, abstraction, localization, hiding, uniformity, completeness, and confirmability. These principles are applied in various combinations throughout the fundamental software life cycle (see Figure 4) to achieve the desired goals discussed above [5,25].

The decomposition of a system [29] depicts the programs or modules of the system organized into a structure by the relationships (interfaces) among them. The seven principles, singly and in combination, are used to determine and control those relationships [26]. They are used as decision criteria to ensure that the resulting decomposition attains the goals of the software system. Thus, each principle deals with some aspect of the relationships-i.e., the interfaces among the modules or programs. The rest of this section discusses each principle separately.

Modularity deals with the properties of a hierarchical software structure. It has been given various definitions by several authors [30-36]. Basically modularity deals with how the structure of an object can make the attainment of some purpose easier. In essence, modularity is purposeful structuring [29]. Therefore, the principle of modularity is made concrete by explaining how certain constraints on the structure of systems can make it easier or harder to achieve some goal such as modifiability, efficiency or reliability.

Imposing constraints on structures is the essence of applying the modularity principle in software engineering [26]. For example, top-down structured programming [36] which forces programmers to make explicit the conditions under which programs are designed and coded can help ensure understandability and prevent errors [26].

It may be possible for a given program to satisfy all goals simultaneously. A program may have one structure if modules are constructed according to one rule (module strength) and a different structure if a different rule (module coupling) is considered [4,37].

Abstraction is a very pervasive principle [34,21]. Despite the existence of the above papers, no practical definition of abstraction exists. However, most researchers in this field agree that the essence of abstraction is to extract essential properties while omitting nonessential details. Hierarchical decomposition in the form of levels shows abstraction in its best form. Each level of the decomposition shows an abstract view of the lower levels purely in the sense that details are subordinated to the lower levels [26]. The top level expresses the program in terms natural to the originator of the task while lower levels express commitments to specific ways of realizing the terms of the higher levels [39].

When combined with the principle of completeness, abstraction ensures that a given level in a decomposition is understandable as a unit without requiring either knowledge of the lower levels of detail, or necessarily how it participates in the system as viewed from a higher level. As such, this principle is employed on the one hand to obtain a description of some level of the system which could be realized by any of several implementations, and on the other hand to give a description of one part of a system which could be used in many other systems requiring the same component at that level of abstraction.

Abstraction interacts strongly with the purpose underlying any particular decomposition. Unless it is combined with the principle of modularity, abstraction is of little practical value. When employed to achieve the goal of understandability, each decomposition level while presenting more and more detailed views of the system must do so in terms that are understandable to the intended user [26].

Localization is concerned with physical proximity. Things must be brought together in one place. Thus, the localization principle deals with physical interfaces, textual sequence, memory, etc. The other principles can interrelate the localized things to serve specific purposes.

Logical and physical records as well as paged memories are examples of localization. Also the avoidance of GOTO's in structured programming is an application of localization to control structures which simplifies confirmability and enhances understandability [26].

The Hiding principle, as discussed by Parnas [29], is used as the major criterion for a decomposition into modules. Although it is not the same, it is related to the idea of postponing binding decisions in top-down programming. The purpose of hiding is to make visible only those properties of a module needed to interface with other modules and to make inaccessable details that should not affect other parts of a system. Abstraction assists in identifying details that should be hidden. Basically, hiding is concerned with access constraints [29].

Uniformity is also an important principle. Since it ensures consistency, it is an obvious principle to apply in software engineering. It is applied to notational matters to yield notation (documentation) that is free of confusing and perhaps costly inconsistencies. When conbined with the abstraction principle, uniformity implies a notation that permits arbitrary mechanization of the internal detailing of an object (the notation does not constrain one's choice of implementation). Also, when the hiding principle is added, the result is a notation that does not permit several implementation choices and also ensures that no unnecessary details of specific implementaion are revealed by the notation. Basically, uniformity is the lack of inconsistencies and unnecessary differences [26].

<u>Completeness</u> is another obviously important principle. This principle ensures that all the essentials of an abstraction are explicit and that nothing essential is left out. Every detail does not have to be shown, but the set of abstract concepts must cover every detail.

When completeness is applied to notational matters, it requires that a notation provides a means for saying everything that one wants to say. When it is combined with abstraction, completeness implies that a notation should be concise, permitting the suppression of invariant details in favor of highlighting the changeable details. Additionally, completeness, when combined with uniformity and abstraction and applied to the goal of efficiency, allows programmers to select different implementation mechanisms to tune a system's performance without having to change the form of any subroutine call [26].

<u>Confirmability</u> is a principle that ensures that information needed to verify correctness has been explicitly stated. This information is used for finding out whether stated goals such as reliability have been achieved.

> "Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or im-

plementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walk-throughs of design, egoless programming [38], and other methods that help to ensure that nothing has been overlooked." [26].

Software Metrics

The result of effective software engineering is the production of a program that meets the requirements (assuming the requirements are accurately stated) of the user. But, how can software be measured so it can be compaired against specified goals of the user? Currently, measures of software attributes seem to be an answer.

The term "metric" by definition means a standard of measure. A software metric is defined as a measure of the extent or degree to which software possesses and exhibits a certain property or attribute [27,28,40]. Software metrics is discussed briefly in the following paragraphs. Chapter IV will concentrate on the metrics applied to COBOL source code as a measure of program composition.

It seems obvious that the software profession is at the point of moving from a handicraft into an engineering industry. There have been enough large failures in software projects to motivate us to acquire full control over the software technology. To be successful and have full control, we must be able to recognize and measure all critical factors, and not simply the easily available ones, such as space and time consumption. Software metrics is concerned with measuring all factors, simple and critical, related to software. In particular, measures relating to the use of human talent resources are of major interest because of its scarcity today, compared to the relatively cheap machine resources. Also, measures related to reliability are becoming more and more important as computers are increasingly used for crucial functions [40]. There are many other software metrics (see Figure 6) such as maintainability, portability, understandability, etc., but this paper is concerned with measuring one characteristic--internal complexity of COBOL programs. These metrics will be discussed in detail in chapter 4.

Summary

There are many aspects of software engineering. The intent of this chapter has been to focus the underlying goals and principles of software engineering into a coherent framework for the readers of this paper. Software metrics is applied to determine to what degree a certain attribute is present in software.

III. SOFTWARE RELIABILITY CONCEPTS

What is Software Reliability?

The most significant problem facing the software professional today is unreliable software. This is the reason for recent emphasis on developing a software reliability theory. As previously defined, software reliability is the probability that a given program operates correctly, without an error, for some time period on the machine for which it was designed. Software reliability is thus a function of the number of errors in a program.

Reliability is not an inherent property of a program; it is largely related to how the program was designed, constructed, tested, and operated. The word probability in the definition actually represents the probability that there are no errors in the program given a valid input from its input space. At times it is simply used as a qualitative measure of the lack of errors in a program [4].

Reliability as a Measure of Software Quality

To provide a meaningful assessment of software quality, quantitative methods of evaluating software are being developed. Until recently, quality assessments have been subjective evaluations of software based on program deficiencies.

However, subjective evaluations for software are not consistent with the use of the methodologies used to measure the quality of hardware. For complex computer systems, consisting of hardware, software, and human interface subsystems, the most meaningful measure of quality is total system reliability. As such, the most meaningful measure of software quality is the reliability of the software subsystem. If software reliability is not explicitly stated it must be determined from the specification of the total system. A study of the total system reliability and cost-benefit trade-offs will determine the reliability apportionment among the hardware, software and human operated subsystems [41].

What is an Error?

Although software reliability is the most appropriate measure of software quality, there are terminology problems because the meanings of such words as software failure and software errors are not entirely obvious by analogy with the corresponding hardware reliability concepts which are well defined. A software error is present when an input is made or a command is given and the program does not respond as the user expects it to. A failure is an occurrence of an error. A failure may be manifested in many ways. A complete stoppage of the program may or may not occur.

Detection of failures is, to a large extent, a subjective decision which must be made by the users or the test personnel. Hopefully, this decision will be made on the basis of objective criteria such as performance specifications. In actual practice, failure detection depends on a user's observation of an error, so, in effect, a software failure is what a user says is an error.

After failures are detected a programmer must analyze the program and locate the causes of the failure. Basically all errors are design or implementation errors. Logical or clerical errors in coding may be found to be responsible for producing the incorrect results. Also the program specification could be in error. When errors are located, action is taken to correct the errors to prevent recurrence of the failures. The correspondence between software errors uncovered and software failures detected is not necessarily one-to-one. Many errors may occur without a failure being detected, and a failure may be a result of several errors. Also, a software failure may be reported that is in fact no software failure at all, but rather a user or hardware deficiency.

Failures differ with respect to their impact on the mission of the software. Severe failures may result in a failure of a mission, while less severe failures may only cause aggravations or limitations which have little effect on the overall mission of the total system.

The reader, if he has written a large program, should now be able to grasp the elusive nature of software reliability. Software errors are not an inherent property of software. Errors are basically human mistakes and we can never expect to find them all--regardless of how well we test the programs. But, we can measure or predict the number of residual errors so we can decide when the software has reached an acceptable reliability level.

Do Software Failures Occur Randomly with Time?

Unlike hardware, there is no physical mechanism which generates software failures. When all errors are removed, the software is 100 per cent reliable and will remain so forever, provided no program changes are made. What then accounts for the randomness of software failures?

Different input combinations result in a different response from the software. The paths traversed within a software program depend on the input combinations. Each path can be thought of as containing possible software errors waiting to be discovered. Without correction, the same errors will occur each time the same logic path is executed. If the errors result in an observable software failure, the given failure can be reproduced at will, or it can be avoided by user control of the input combinations. Therefore, software failures are functions of the input combinations--not random functions of time. However, in reality, input combinations are chosen in a somewhat random fashion, and the resultant effect is that errors are uncovered and failures are observed at random. It is with this meaning that we talk about the random occurrence of software failures [41].

Reliability Models

The most important unknown of software reliability is the number of residual errors in a program. If an estimate of this number were available during the testing stages it would help determine when to stop testing. Also if we knew the number of remaining errors in an operational program we could estimate the cost of maintenance and establish a level of confidence in the program. Other related attributes for which estimates are desirable are the reliability of the program and the mean-time-to-failure of the program. Measures of the program's complexity would be useful to estimate the number of errors and to judge the quality of the design. If software reliability models were available that would model software failures, then one could deal with the unknowns of software reliability [4].

There are 3 types of software reliability models being evolved today. A number of software reliability models are discussed in references [42-47]. These models are closely related to hardware reliability theory and contain significant assumptions about the underlying probability distribution of software failures. References [48-56] are reliability studies which contain evaluations of these models relative to specific error data. These models seem to apply only to specific situations and do not have general application in any environment. The next set of models, discussed in references [11,57-59], produce similar results, but are not based on hardware reliability theory. The last set of models is concerned with predicting the complexity of a program [14-22]. The rest of this chapter will discuss the hazard function for software failures.

Hazard Function for Software Failures

We shall assume that a large program is resident in a computer and is servicing a steady stream of dissimilar "inputs". We shall assume that these inputs enter the program at arbitrary points in time, and that each such entry can be looked upon as an opportunity to detect an error in the program. Thus, we assume that software errors are detected in a random manner.

Software does not age with time, therefore, it is reasonable to assume that its failure rate is constant between points in time at which changes are made. Every time an error is detected, we eliminate it. If we ignore the possibility of introducing new errors then our failure rate is a step function as indicated in Figure 7. Several variations and justifications for this model have appeared in the literature. For the purpose of this paper, it will suffice to illustrate that a program failure rate is decreasing and will eventually go to zero, assuming there are no modifications for new capabilities. This is in contrast to the hazard function of a hardware component (see Figure 8).

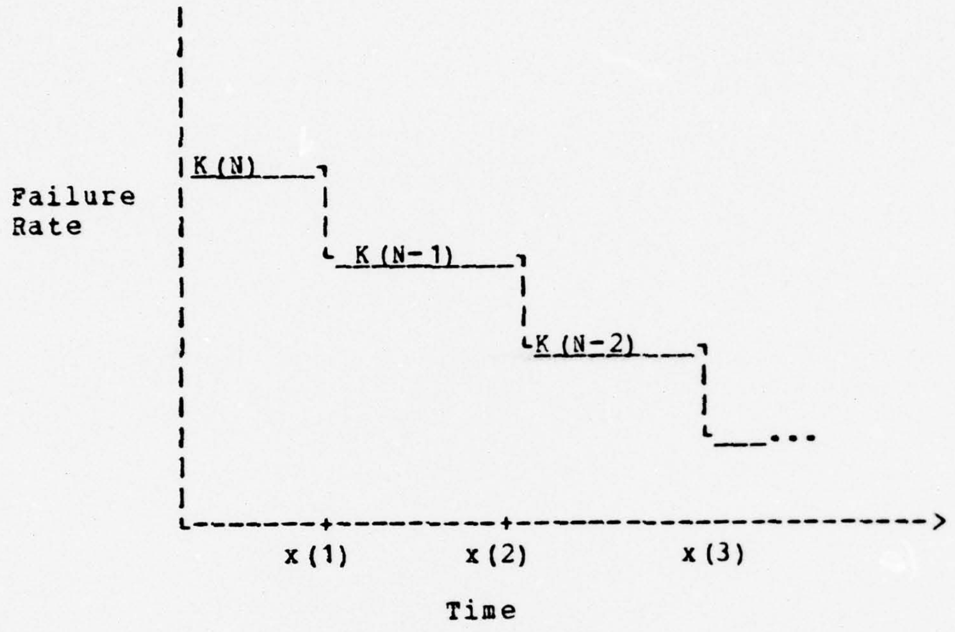


Figure 7. Failure Rate Changes As Errors Are Removed

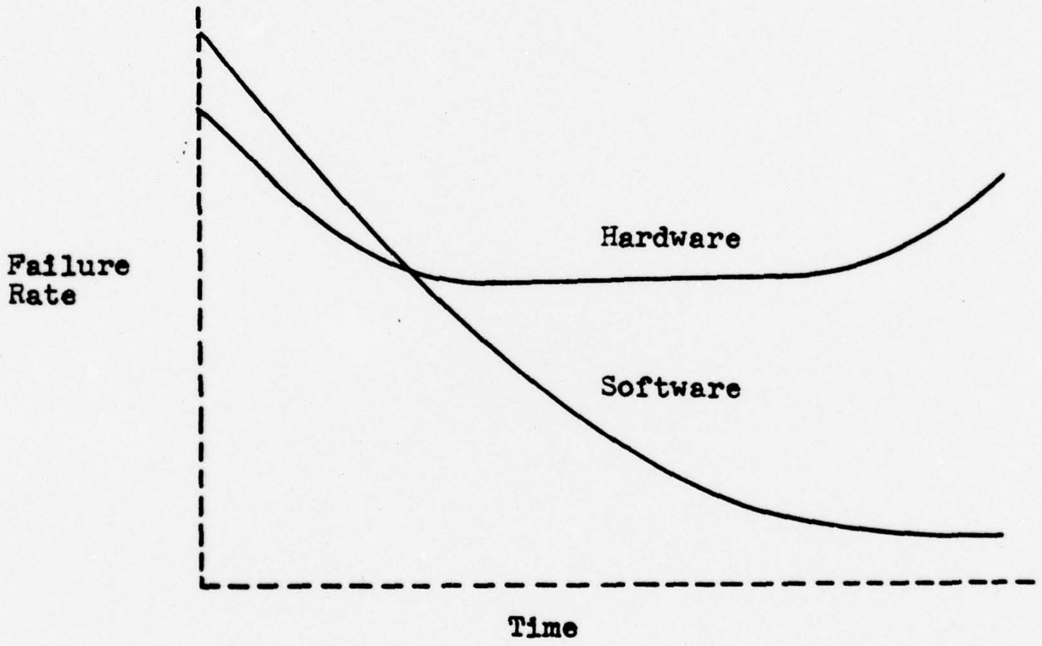


Figure 8. Hardware vs. Software Failure Curves

At the start of the testing process, we assume that the program contains an unknown number of errors, say N. The failure rate is assumed to be proportional to the residual number of errors in the program. Every time an error is encountered, the error is removed and no new error is introduced. Although these assumptions make the model less than realistic, Jelinski and Moranda [42] have demonstrated the usefulness of a model of this type in analyzing error data from the U.S. Navy and the Apollo program of NASA.

Let $x(1), x(2), \ldots,$ denote the points in time at which software errors are detected and corrected. Then, according to the assumptions of the model, the failure rate between $x(i-1)$ and $x(i)$ is $K(N-i+1)$, $i = 1,2, \ldots, n$, for $n \leq N$, where K is the constant of proportionality. The failure rate generated by the testing process is illustrated in Figure 7. If $T(i)$ denotes the time interval between $x(i-1)$ and $x(i)$, then from the assumption that time-to-failure is exponentially distributed

$F[t(i)] = P[T(i) \leq t(i)] = 1 - \exp[-K(N-i+1)t(i)]$, $K > 0$, and $t(i) \geq 0$.

If K and N were known, then the reliability of the software prior to testing, the distribution function of the time to test, the number of errors to be removed in order to obtain a desired level of reliability, and other such information can easily be determined. In practice, K and N are unknown, and hence the estimation of K and N becomes critical.

In order to estimate K and N, and to obtain a stopping rule for testing the program, one must use t(1), t(2), ... , t(n) as the realizations of T(1), T(2), ... , T(n), for $n \leq N$. This estimation problem is an unusual one. The time intervals t(1), t(2), ... , t(n), do not constitute a random sample of size n from a single failure distribution, but rather n samples, each of size 1, from n different but related distributions [10]. Therein lies the problem of predicting K and N.

Schneidewind [60,61] showed that error data fitted no single underlying probability distribution. This is more reason to believe that failure functions are not only functions of the remaining errors but also the composition of the program itself.

The remaining chapters of this paper will be devoted to developing and discussing a model for estimating the number of errors in COBOL programs.

IV. DESCRIPTION OF DATA

Introduction

Error data processing involves three interrelated activities: collection, classification into error categories, and analysis. Since collection and classification have been dealt with adequately in other sources [61-67] [50] [56], analysis is the primary concern of this paper. While analysis has been the primary concern of this research, the other two were considered by using concepts and techniques developed in other studies.

The purpose of this chapter is to describe data that are used to develop a model for predicting the number of errors in COBOL programs. Chapter V will present a detailed analysis of the data described in this chapter.

Terminclogy Revisited

Although defined elsewhere, several terms need clarification to familarize the reader with what follows in the rest of this paper.

The term software reliability, for the purpose of the analysis of empirical data, needs to be redefined. Software posseses reliability to the extent that it is expected to perform its intended functions satisfactorily. With this in mind, errors in programs represent an inability of the programs to perform intended functions satisfactorily. An error-free program would be a reliable program. Henceforth in this paper, anything that causes software not to perform its intended function is an error. Specifically, the term error is a user dissatisfaction, which is documented on a form, with the results of a program. The error may not necessarily be the result of an execution of a program, e.g., design reviews can result in the detection of errors.

The term project is the combination of development activities required to produce the software and its documentation. Three sources of data are used in this report. Because of the restrictions in employing the actual system and program names, the data sources are called Project 1, Project 2, and Project 3. Each one will be discussed later.

Project Descriptions

The three projects represent small to large software development activities. The application software for all three projects is written in COBOL. The smallest compilable unit of source code is the program. Each project is discussed below. Table 1 lists the data available from each project.

Project 1

Project 1 is a data collection system [67] consisting of 5 batch programs with a total of 2280 lines of code. The system provides an on-going data base for input into re-

TABLE 1. Data Availability for Each Project

	Project 1	Project 2	Project 3
1) General Project Descriptions	X	X	X
2) Design Problem Data	X	X	
3) Problem Report (Error) Data	X	X	X
4) Software Characteristics	X	X	X
5) Testing Data	X	X	
6) Computer Usage Data	X	X	

liability models. The data base also contains program characteristics as discussed in this paper. The system applies to COBOL programs designed to execute on the Honeywell H6060 computer system throughout the Air Force. The 5 programs in this system utilize a file management system available on the H6060.

Project 2

Project 2 is an on-line system involving several kinds of data processing activities such as personnel management, accounting and finance, inventory etc. Only 14 programs are available for analysis. There are 19045 lines of source code in these programs. These programs execute on the National Cash Register NCR8200 computer system.

Project 3

Project 3 represents an initial delivery of a large on-line Command Manpower Data System(CMDS). CMDS is a resource accounting and management information system which supports the Manpower and Organization function at Major Command level throughout the Air Force. Data for 46 programs are available for analysis. There are 54116 lines of source code in these programs. These programs execute on the H6060 computer system and perform a wide variety of data processing activities, general purpose utility, data retrieval, data maintenance, etc.

Approach to Data Collection and Classification

It would be ideal to perform a study of this nature using the same collection and classification tools and procedures for all projects. Since real data from on-going projects within different organizations are being used, this was not possible. Data sources are the normal data collection and classification system of the organization developing the software. For example, the Air Force Data System Design Center(AFDSDC) has a manual system for collecting error reports. Project 3 data was recorded using this system.

Although the data is reasonably good, it is obvious that it is not the same type of data from all projects. This presented a problem when trying to classify an error according to a specific category. Finally it was decided to work only with actual errors that required a change in source code to affect corrective action. By considering only code change errors and performing analysis at the individual program level, it was possible to generate similiar data from all projects. Errors were classified into 12 categories. These categories are:

1) Computationa,
2) Logic,
3) Data Input,
4) Data Handling,
5) Data Output,
6) Interface,

7) Array Processing,

8) Data Base,

9) Operation,

10) Program Execution,

11) Documentation, and

12) Other

The categories along with types of errors in each category are presented in Table 2.

Software Characteristics

Boehm [27] presents a detailed discussion of characteristics of software quality (see Figure 6). Thayer and Lipow [50] discuss the two forms of software quality characteristics, those that can be quantitatively measured and those that require some subjective evaluation. Both are needed to explain errors. Both forms were considered by Thayer and Lipow and examples are presented in Table 3. The subjective form did not show much promise as predictor variables for the number of errors in programs. Since previous research showed that software structure influenced the number of errors and since our primary objective is to develop a complexity model for predicting the number of errors in COBOL programs, this paper is concerned with only structural characteristics. Only those that can be measured are considered in this report.

TABLE 2. Error Categories

COMPUTATIONAL ERRORS

Incorrect operand in equation
Incorrect use of parenthesis
Sign convention error
Units or data conversion error
Computation produces an over/under flow
Incorrect/inaccurate equation used
Precision loss due to mixed mode
Missing computation
Rounding or truncation error

LOGIC ERRORS

Incorrect operand in logical expression
Logic activities out of sequence
Wrong variable being checked
Missing logic or condition tests
Too many/few statements in loop
Loop iterated incorrect number of times
(including endless loop)
Duplicate logic

DATA INPUT ERRORS

Invalid input read from correct data file
Input read from incorrect data file
Incorrect input format
Incorrect format statement referenced
End of file encountered prematurely
End of file missing

DATA HANDLING ERRORS

Data file not rewound before reading
Data initialization not done
Data initialization done improperly
Variable used as a flag or index not set properly
Variable referred to by the wrong name
Bit manipulation done incorrectly
Incorrect variable type
Data packing/unpacking error
Sort error

TABLE 2. Error Categories (Continued)

DATA OUTPUT ERRORS

Data written on wrong file
Data written according to the wrong format statement
Data written in wrong format
Data written with wrong carriage control
Incomplete or missing output
Output field size too small
Line count or page eject problem
Output garbled or misleading

INTERFACE ERRORS

Wrong subroutine called
Call to subroutine not made or made in wrong place
Subroutine arguments not consistent in type, units, order, etc.
Subroutine called is nonexistent
Software/data base interface error
Software user interface error
Software/software interface error

ARRAY PROCESSING ERRORS

Data not properly defined/dimensioned
Data referenced out of bounds
Data being referenced at incorrect location
Data pointers not incremented properly

DATA BASE ERRORS

Data not initialized in data base
Data initialized to incorrect value
Data units are incorrect

OPERATION ERRORS

Operating system error (vendor supplied)
Hardware error
Operator error
Test execution error
User misunderstanding/error
Configuration control error

PROGRAM EXECUTION ERROR

Bad object code

TABLE 2. Error Categories (Continued)

DOCUMENTATION ERRORS

User manual
Interface specification
Design specification
Requirements specification
Test documentation

OTHER

Time limit exceeded
Core storage limit exceeded
Output line limit exceeded
Compilation error
Code or design inefficient/not necessary
User/programmer requested enhancement
Design nonresponsive to requirements
Code delivery or redelivery
Software not compatible with project standing

TABLE 3. Available Parameters

<u>Program Structural Characteristics</u>

- Program size
  - Total source code statements
  - Executable statements
  - Non-executable statements
  - Machine dependent number of instructions (ENTER SYMBOLIC)
- Number of unconditional branches
- Number of conditions in program
- Number of direct interfaces
  - With routines within program and other application programs
  - With operating system
- Number of arguments in interface calls
- Data interfaces
  - Number of global data blocks
  - Number of internal data variables
- Number of procedures
- Number of entry points
- Number of exit points
- Routine code type
  - Number of computational
  - Number of logical
  - Number of data handling
  - Number of I/O
- Loop and nesting levels
- Pages of documentation

TABLE 3. Available Parameters (Continued)

Computer time (clock time, not CPU time)

- Development time
- Test time

Subjective Characteristics

Routine difficulty at preliminary design

Routine difficulty after formal test and delivery

- Design
- Code
- Debug/checkout
- Implementation
- Documentation

Routine type

- Executive
- Control
- Setup
- Input
- Computational
- Post processing
- Output

Personnel data

- Number of people working on routine
- Load factor on each programmer
- Programmer rating
- Programmer/job evaluation

Structural Characteristics

Program structural characteristics are measurable. They quantify the actual physical attributes of a program. The application of metrics allows the quantification of such things as a program's size, input/output patterns, use of a data base, computations performed, interfaces, use of the various language elements, and logical complexity [17,18].

The approach taken was to provide as much quantative detail as possible. In an effort to tie specific error categories to types of code within a program, 22 generic types of structural characteristics were chosen as language metrics. The structural characteristics chosen for this study are presented in Table 4. Please note that a measure for each error category is included. The purpose for choosing these characteristics is to measure the likelihood that a program may have particular kinds of errors. These characterisitcs will also be useful in future studies of error type distributions. Since there were no automated tools available to collect structure data, a program was developed by the author to analyze COBOL source code. This program is called COBOL Characteristics Analyzer Program (CCA). It was originally designed for the NCR8200 computer system, and has been converted to run on the H6060 computer.

TABLE 4. STRUCTURAL CHARACTERISTICS DEFINITIONS

Metric Variable	Definitions
LC	Number of logical conditions
IO	Number of input/output statements
CO	Number of arithmetic statements
DH	Number of data transfer statements
PC	Number of CALLS to external and internal routines
UBR	Number of unconditional branches
EXIT	Number of EXIT statements
STOP	Number of STOP statements
OSC	Number of CALLS to operating system
CC	Number of CALLS to compiler to COPY source code from the library
TS	Total statements = NEX + NNEX
NEX	Number of executable statements
NNEX	Number of non-executable statements
FD	Number of file descriptions
RD	Number of record descriptions or "01" level descriptions
DD	Number of data item descriptions
TD	Total descriptions = FD + RD + DD
DR	Number of data references
NCO	Number of comments

TABLE 4. STRUCTURAL CHARACTERISTICS DEFINITIONS
continued

Metric Variable	Definitions
PAR	Number of paragraphs
NL	Number of source lines There can be more than one statement per line.
RW	Number of references to "reserved" words

COBOL Characteristics Analyzer Program

CCA is a utility program which statistically analyzes COBOL source. It breaks a program's code into its language elements. This analysis is done at the program level; however, it identifies interfaces between routines, between the subject program and other application programs, and between the subject program and the operating system. Table 4 has presented the list of metrics chosen to quantify the structural characteristics of COBOL programs. CCA computes the values for these metrics. Figure 9 presents sample output for a program called S-PTU0.

Please note that the columns PERCENT OF TOTAL and PERCENT OF EXECUTABLE (see Figure 9) require special interpretation. For example, the number of logical is 80. This is not the number of logical statements in the program, it represents the number of logical conditions in the program.

```
                                        PROGRAM CHARACTERISTICS SUMMARY
                                                     FOR
                                                   S-PTUO
   DATE: 01  25, 1977                                                                     VERSION DATE: 01  25,1977

TOTAL STATEMENTS=    1168          NUMBER EXECUTABLE STATEMENTS=    566          NUMBER NON EXECUTABLE STATEMENTS=    602

NUMBER FILE DESCR=      6          NUMBER OF RECORD DESCRIPTIONS=    34          NUMBER OF DATA ITEM DESCRIPTIONS=    536

TOTAL DECLARATIONS=   576          NUMBER OF DATA REFERENCES=      4447          NUMBER OF COMMENTS=                  613

NUMBER PARAGRAPHS=     60          NUMBER OF LINES=                2040          NUMBER RESERVE WORDS=                562

STATEMENT PROFILES                  NUMRER         PERCENT OF    PERCENT OF
                                                      TOTAL      EXECUTABLE

***     LOGICAL                       80              6.84         14.13
***     INPUT/OUTPUT                 185             15.83         32.68
***     COMPUTATIONAL                118             10.10         20.84
***     DATA HANDLING                100              8.56         17.66
***     PROCEDURE CALLS               27              2.31          4.77
***     UNCONDITIONAL BRANCHES       139             11.90         24.55
***     EXIT                          12              1.02          2.12
***     STOP                           3               .25           .53
***     OPERATING SYSTEM CALLS
***     COMPILER CALLS                15              1.28          2.65
```

Figure 9. Example Output from CCA.

Each AND and OR is counted as a logical condition. Because of this the PERCENT columns will not add to 100 percent.

Summary of Available Data for Each Project

Individual project data is summarized in Tables 5-7 respectively. Tables 8-10 contain descriptive statistics for individual project data. The error data were collected from software discrepancy reports provided by project programmers. Program CCA was used to collect the structural characteristics data. This data is analyzed in the next chapter. When, in the course of analysis, specific project data are germane to results, the reader is encouraged to refer back to the corresponding data.

TABLE 5. PROJECT ONE DATA

Program No.	Total No. Of Errors (N)	METRIC VARIABLES BY PROGRAM																					
		LC	I/O	CO	DH	PC	UBR	EXIT	STOP	OSC	CC	TS	NEX	NNEX	FD	RD	DD	TD	DR	NCO	PAR	NL	RW
1	25	43	32	15	74	15	26	12	2	15	4	556	223	333	5	21	174	200	581	0	43	641	217
2	13	23	7	1	6	5	12	3	1	7	2	95	62	33	1	2	28	31	152	0	12	117	45
3	68	119	20	26	146	16	67	12	1	5	0	598	396	202	4	25	131	160	896	2	42	597	373
4	15	18	16	8	29	12	18	10	1	3	0	183	103	80	5	7	62	74	255	16	27	254	107
5	58	77	31	55	131	18	56	10	1	4	4	619	369	250	5	29	208	242	990	32	36	673	450

TABLE 6. PROJECT TWO DATA

Program No.	Total No. Of Errors (N)	METRIC VARIABLES BY PROGRAM																					
		LC	I/O	CO	DH	PC	UBR	EXIT	STOP	OSC	CC	TS	NEX	NNEX	FD	RD	DD	TD	DR	NCO	PAR	NL	RW
1	12	21	36	51	94	16	13	7	1	∅	∅	423	219	204	3	25	161	197	1560	49	24	544	286
2	9	18	29	∅	2∅	7	22	1	1	∅	∅	114	8∅	34	2	9	17	28	484	25	7	176	76
3	17	34	103	25	126	10	30	5	1	∅	∅	664	3∅2	362	6	30	3∅8	345	2350	250	23	1053	361
4	22	59	57	53	124	18	59	9	1	∅	∅	519	317	2∅2	5	26	159	19∅	12∅6	1∅9	22	723	326
5	5	9	18	1	21	2	3	∅	1	∅	∅	92	52	4∅	3	5	22	3∅	1∅2	37	8	154	45
6	∅	2	16	5	2∅	2	3	1	1	∅	∅	127	48	79	3	7	6∅	7∅	94	43	8	2∅6	4∅
7	5	11	8∅	5	81	3∅	37	11	1	∅	∅	661	2∅6	455	9	51	349	4∅9	1128	265	25	1051	2∅4
8	9	26	57	16	57	17	24	1∅	1	∅	∅	286	174	112	5	16	77	98	11∅∅	115	23	498	224
9	4	11	65	8	27	5	26	4	1	∅	∅	377	123	254	5	16	216	237	681	114	14	584	113
10	84	255	238	36	548	182	642	7∅	2	∅	3	2125	1420	7∅5	4	58	629	691	7641	642	256	3566	1660
11	58	18∅	167	139	341	72	69	47	1	∅	3	1898	912	986	5	5∅	915	97∅	6062	575	76	274∅	1041
12	32	1∅3	111	51	289	132	62	26	1	∅	∅	2345	621	1724	4	118	1592	1714	3853	353	58	3∅12	806
13	45	8∅	185	118	1∅∅	27	139	12	3	∅	15	1168	566	6∅2	6	34	536	576	4447	613	6∅	2∅4∅	562
14	2∅	36	169	21	492	25	63	71	2	∅	∅	2∅28	81∅	1218	5	134	1∅63	12∅2	4466	516	44	2688	741

TABLE 7. PROJECT THREE DATA

Program No.	Total No. Of Errors (N)	METRIC VARIABLES BY PROGRAMS																				
		LC	I/O	CO	DH	PC	UBR	EXIT	STOP	OSC	CC	TS	NEX	NNEX	FD	RD	DD	TD	DR	NCO	PAR	NL	RW
1	25	512	202	114	489	134	386	31	1	25	2	3392	1896	1496	8	139	1253	1400	3825	17	228	3421	2220
2	21	119	139	38	193	46	140	17	1	12	2	1384	698	686	9	71	558	633	1388	11	106	1494	799
3	18	3	12	0	0	6	0	1	0	2	0	52	27	25	3	5	7	15	47	0	6	89	25
4	24	433	76	47	405	175	234	14	1	55	2	1820	1442	378	8	37	291	336	3165	6	130	1624	1557
5	3	27	38	16	73	11	30	4	1	8	2	487	215	272	2	25	151	173	490	1	24	511	263
6	5	33	39	16	81	11	32	4	1	9	2	502	228	272	2	25	153	180	533	1	24	525	239
7	8	157	12	57	165	28	53	3	1	33	2	747	511	236	2	46	181	229	1163	2	33	676	534
8	23	733	48	116	686	155	194	15	1	79	3	3298	2030	1268	2	112	1131	1249	4731	29	119	2925	2144
9	4	59	40	13	86	25	17	1	1	4	0	635	246	389	2	39	302	343	639	0	22	649	289
10	2	18	36	4	47	14	8	4	1	9	0	419	141	278	2	28	224	254	357	0	20	454	166
11	1	3	12	3	14	3	2	0	1	7	0	125	45	80	1	12	59	72	107	0	7	155	57
12	67	1584	54	233	1314	338	1012	20	1	41	2	5323	4599	724	5	46	656	707	11533	45	373	4162	3465
13	5	92	13	26	195	68	59	3	1	4	2	777	463	314	2	27	272	301	1025	9	44	771	345
14	3	15	18	0	20	0	10	3	1	1	0	187	68	119	3	9	104	116	143	0	12	246	56
15	5	81	77	13	191	42	75	2	0	1	2	902	484	418	4	45	336	385	966	3	44	858	437
16	3	59	58	11	59	17	36	4	1	18	4	534	269	265	6	48	193	247	622	20	29	609	294
17	24	597	145	104	440	79	480	18	1	27	2	2402	1893	509	10	56	415	481	4090	122	205	2645	1990
18	30	749	81	59	926	991	417	76	1	22	0	4009	3322	639	6	62	574	642	7572	0	432	3167	2756
19	19	227	74	35	266	60	133	11	1	91	2	1287	900	387	5	29	341	375	2169	16	92	1233	683

TABLE 7. PROJECT THREE DATA
Continued

Program No.	Total No. Of Errors (N)	METRIC VARIABLES BY PROGRAMS																				
		LC	I/O	CO	DH	PC	UBR	EXIT	STOP	OSC	CC	TS	NEX	NNEX	FD	RD	DD	TD	DR	NCO	PAR	NL	RW
20	12	193	117	18	1Ø5	19	92	5	1	11	2	881	563	318	6	38	242	286	1439	Ø	44	832	676
21	17	227	195	91	365	7Ø	173	18	1	49	2	1675	1191	484	4	4Ø	4Ø8	452	2953	34	151	178Ø	1274
22	5	94	87	48	145	54	46	8	1	9	2	732	494	238	3	29	186	218	152Ø	Ø	61	731	487
23	3	57	54	13	1Ø5	3Ø	33	2	1	8	2	448	3Ø5	143	3	17	11Ø	13Ø	789	Ø	36	431	321
24	4	82	48	19	136	46	66	4	1	14	2	6Ø3	418	185	3	26	139	168	1181	47	68	744	436
25	6	132	15Ø	4Ø	272	45	74	3	2	11	2	988	731	257	4	29	2Ø7	24Ø	218Ø	Ø	74	969	794
26	7	132	15Ø																2178	Ø			
27	16	1Ø4	87	66	26Ø	43	66	4	2	11	2	9Ø7	645	262	4	3Ø	2Ø7	241	1919	1	72	932	8Ø3
28	8	74	9Ø	16	245	1Ø	69	6	1	25	2	1Ø73	538	535	14	48	42Ø	482	1283	58	4Ø	1253	523
29	14	263	123	41	357	77	217	2Ø	1	14	2	1638	112Ø	518	8	42	44Ø	49Ø	2840	38	147	1995	1Ø76
30	15	129	1Ø3	32	449	81	86	8	1	11	2	1655	9Ø2	753	5	49	664	718	2161	139	86	1961	913
31	3	141	8Ø	25	262	35	96	8	2	23	2	1272	675	597	3	35	532	57Ø	1486	1Ø4	62	1435	1ØØØ
32	5	62	12	6	69	12	43	4	1	48	2	4Ø5	259	146	4	18	111	133	635	3Ø	24	489	339
33	13	149	51	5	369	64	39	7	1	8	2	14Ø5	745	66Ø	8	41	572	621	16ØØ	211	63	1682	814
34	7	67	65	1Ø	146	9	43	4	1	21	3	7Ø9	368	341	4	35	277	316	1126	74	4Ø	937	6Ø4
35	15	223	57	39	363	5Ø	163	11	1	33	2	144	942	5Ø2	4	32	447	483	2Ø8Ø	19	95	1488	811
36	8	121	8Ø	77	476	77	195	6	1	6	2	163Ø	1Ø41	589	4	46	499	549	2319	1	164	184Ø	1Ø97
37	11	163	36	36	197	39	1Ø2	1Ø	1	37	2	915	623	292	8	41	215	264	1693	39	97	1Ø85	6Ø3
38	22	258	48	5Ø	356	27	21Ø	9	Ø	57	Ø	1195	1Ø15	18Ø	2	13	164	179	2275	6	93	1Ø8Ø	854

TABLE 7. PROJECT THREE DATA
Continued

Program No.	Total No. Of Errors (N)	METRIC VARIABLES BY PROGRAMS																				
		LC	I/O	CO	DH	PC	UBR	EXIT	STOP	OSC	CC	TS	NEX	NNEX	FD	RD	DD	TD	DR	NCO	PAR	NL	RW
39	9	20	26	3	14	0	26	0	1	4	2	153	96	57	3	10	33	46	209	28	15	217	90
40	6	114	33	25	234	23	55	12	1	16	2	913	515	398	4	56	322	382	1382	26	49	977	502
41	2	29	41	15	87	16	18	6	1	9	2	416	224	192	3	20	152	175	577	41	34	534	237
42	1	15	18	3	3	12	4	1	1	25	2	137	84	53	1	4	44	49	279	0	9	173	76
43	5	54	21	15	97	18	43	3	2	20	2	432	275	157	2	15	122	139	633	1	28	448	273
44	12	222	41	20	261	20	194	2	1	49	2	1555	810	745	5	71	651	727	2080	20	68	1705	1240
45	18	13	30	14	105	25	40	1	1	0	0	514	289	225	3	17	193	213	702	0	35	555	335
46	4	86	42	31	116	26	82	9	1.	30	2	640	425	215	4	29	167	200	912	2	77	735	418

TABLE 8. DESCRIPTIVE STATISTICS FOR PROJECT ONE DATA

VARIABLE	MEAN	STANDARD DEVIATION	MINIMUM VALUE	MAXIMUM VALUE	STD ERROR OF MEAN	SUM	VARIANCE
N	35.80000000	25.48921340	13.00000000	68.00000000	11.39912277	179.0000000	649.70000
LC	56.00000000	42.16633728	18.00000000	119.00000000	18.85735931	280.0000000	1778.00000
IO	21.20000000	10.52140675	7.00000000	32.00000000	4.70531614	106.0000000	110.70000
CC	21.00000000	21.13054661	1.00000000	55.00000000	9.44986772	105.0000000	446.50000
CH	77.20000000	61.30008157	6.00000000	146.00000000	27.41422988	386.0000000	3757.70000
PC	13.20000000	5.06951674	5.00000000	18.00000000	2.26715681	66.0000000	25.70000
UBR	35.80000000	24.29403219	12.00000000	67.00000000	10.86462148	179.0000000	590.20000
EXIT	9.40000000	3.71483512	3.00000000	12.00000000	1.66132477	47.0000000	13.80000
STCP	1.20000000	0.44721360	1.00000000	2.00000000	0.20000000	6.0000000	0.20000
OSC	6.80000000	4.81663783	3.00000000	15.00000000	2.15406592	34.0000000	23.20000
CC	2.00000000	2.00000000	0.00000000	4.00000000	0.89442719	10.0000000	4.00000
TS	410.20000000	250.54680202	95.00000000	619.00000000	112.04793617	2051.0000000	62773.70000
NEX	230.60000000	151.06058387	62.00000000	396.00000000	67.55634685	1153.0000000	22819.30000
NNEX	179.60000000	122.88327795	33.00000000	333.00000000	54.95537256	898.0000000	15100.30000
FC	4.00000000	1.73205081	1.00000000	5.00000000	0.77459667	20.0000000	3.00000
RC	16.80000000	11.71324037	2.00000000	29.00000000	5.23832034	84.0000000	137.20000
CC	120.60000000	75.17845436	28.00000000	208.00000000	33.62082688	603.0000000	5651.80000
TC	141.40000000	87.50885669	31.00000000	242.00000000	39.13515044	707.0000000	7657.80000
DR	574.80000000	373.03846987	152.00000000	990.00000000	166.82787537	2874.0000000	139157.70000
NCC	10.00000000	14.00000000	0.00000000	32.00000000	6.26099034	50.0000000	196.00000
PAR	32.00000000	12.86468033	12.00000000	43.00000000	5.75325995	160.0000000	165.50000
NL	456.40000000	253.43598797	117.00000000	673.00000000	113.34001941	2282.0000000	64229.80000
RW	238.40000000	171.76961315	45.00000000	450.00000000	76.81770629	1192.0000000	29504.80000

TABLE 9. DESCRIPTIVE STATISTICS FOR PROJECT TWO DATA

VARIABLE	MEAN	STANDARD DEVIATION	MINIMUM VALUE	MAXIMUM VALUE	STD ERROR OF MEAN	SUM	VARIANCE
N	23.00000000	24.19154334	0.00000000	84.00000000	6.46546192	322.000000	585.2308
LC	60.35714286	73.97151962	2.00000000	255.00000000	19.76972020	845.000000	5471.7857
IC	95.07142857	69.91694445	16.00000000	238.00000000	18.68608940	1331.000000	4888.3791
CC	37.78571429	42.96721214	0.00000000	139.00000000	11.48347048	529.000000	1846.1813
CH	167.14285714	178.22278933	20.00000000	548.00000000	47.63204401	2340.000000	31763.3626
PC	38.92857143	53.93794766	2.00000000	182.00000000	14.41552288	545.000000	2909.3022
UBR	85.14285714	164.13917575	3.00000000	642.00000000	43.86814103	1192.000000	26941.6703
EXIT	19.57142857	24.80916172	0.00000000	71.00000000	6.63052737	274.000000	615.4945
STCP	1.28571429	0.61124985	1.00000000	3.00000000	0.16336339	18.000000	0.3736
OSC	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.000000	0.0000
CC	1.50000000	4.03351345	0.00000000	15.00000000	1.07800181	21.000000	16.2692
TS	916.21428571	828.78049229	92.00000000	2345.00000000	221.50090364	12827.000000	686877.1044
NEX	417.85714286	402.36605170	48.00000000	1420.00000000	107.53685068	5850.000000	161898.4396
NNEX	498.35714286	505.11485474	34.00000000	1724.00000000	134.99762339	6977.000000	255141.0165
FC	4.64285714	1.73680267	2.00000000	9.00000000	0.46418004	65.000000	3.0165
RC	41.35714286	39.63605030	5.00000000	134.00000000	10.59318003	579.000000	1571.0165
CC	436.00000000	468.06278343	17.00000000	1592.00000000	125.09504079	6104.000000	219082.7692
TC	482.64285714	504.22150369	28.00000000	1714.00000000	134.75886511	6757.000000	254239.3242
CR	2512.42857143	2384.84499462	94.00000000	7641.00000000	637.37663503	35174.000000	5687485.6484
NCO	264.71428571	232.95144049	25.00000000	642.00000000	62.25889129	3706.000000	54266.3736
PAR	46.28571429	64.05217653	7.00000000	256.00000000	17.11866425	648.000000	4102.6813
NL	1360.35714286	1193.18143614	154.00000000	3566.00000000	318.89115245	19045.000000	1423661.9396
RW	398.92857143	436.03590725	40.00000000	1660.00000000	117.07092057	5585.000000	191875.4560

BEST AVAILABLE COPY

TABLE 10. DESCRIPTIVE STATISTICS FOR PROJECT THREE DATA

VARIABLE	MEAN	STANDARD DEVIATION	MINIMUM VALUE	MAXIMUM VALUE	STD ERROR OF MEAN	SUM	VARIANCE
N	11.60869565	11.42021261	0.00000000	67.0000000	1.68381849	534.000000	130.4213
LC	191.08695652	274.18329847	3.00000000	1584.0000000	40.42612187	8790.000000	75176.4812
IC	66.30434783	47.53852887	12.00000000	202.0000000	7.00923267	3050.000000	2259.9498
CC	37.02173913	41.74711654	0.00000000	233.0000000	6.15527653	1703.000000	1742.8217
DM	250.45652174	245.88702575	0.00000000	1314.0000000	36.25406466	11521.000000	60460.4314
PC	69.04347826	150.48336291	0.00000000	991.0000000	22.18756140	3176.000000	22645.2425
UBR	124.28260870	171.55946584	0.00000000	1012.0000000	25.29506383	5717.000000	29432.6517
EXIT	9.67391304	13.10649432	0.00000000	76.0000000	1.93244716	445.000000	171.7802
STCP	1.04347826	0.41933925	0.00000000	2.0000000	0.06182820	48.000000	0.1758
GSC	21.91304348	20.42637303	0.00000000	91.0000000	3.01170440	1008.000000	417.2367
CC'	1.71739130	0.86056941	0.00000000	4.0000000	0.12688404	79.000000	0.7406
TS	1165.32608696	1053.88871549	52.00000000	5323.0000000	155.38741378	53605.000000	1110681.4246
NEX	771.76086957	849.20295139	27.00000000	4599.0000000	125.20814433	35501.000000	721145.6527
NNEX	393.50000000	291.75512944	25.00000000	1496.0000000	43.01694700	18101.000000	85121.0556
FC	6.04347826	11.43864118	1.00000000	80.0000000	1.68653563	278.000000	130.8425
RC	37.28260870	25.19846825	3.00000000	139.0000000	3.71531145	1715.000000	634.9628
CC	324.60869565	257.79470542	7.00000000	1253.0000000	38.00975565	14932.000000	66458.1101
TC	367.15217391	281.30931709	45.00000000	1400.0000000	41.47679600	16889.000000	79134.9319
CR	1854.58695652	2039.71842911	47.00000000	11533.0000000	300.74007519	85311.000000	4160451.2700
NCC	26.10869565	42.75627479	0.00000000	211.0000000	6.30406879	1201.000000	1828.0990
PAR	82.19565217	86.43457738	6.00000000	432.0000000	12.74414214	3781.000000	7471.0053
NL	1176.43478261	908.66213382	85.00000000	4162.0000000	133.97492249	54116.000000	825666.8734
RW	775.65217391	727.96682517	25.00000000	3465.0000000	107.33285272	35680.000000	529935.6986

V. ERROR DATA ANALYSIS

Introduction

Error data analysis is important because of the necessity to cope with problems of cost and software unreliability. Examples of areas which benefit from error data analysis include management of software development efforts, design of software engineering techniques and tools, and determining which software characteristics are relevant to software reliability. This chapter is mainly concerned with the latter. The principal emphasis of this analysis has been on individual program error data collected during testing and operational usage.

There are many kinds of data available from the software life cycle (see Figure 5). Since the main emphasis of this research stems from the idea that much can be said about the quality and reliability of software from the software's error history, only error data were analyzed. The primary approach has been not to repeat other research, but to verify and expand previous findings.

The purpose of this chapter is to summarize an analysis of data collected from the three projects. This will be accomplished by presenting the results of an empirical and regression analysis of the raw data. Chapter VI will present the final empirical models developed from the analysis summarized in this chapter.

Analysis of Empirical Data

This section contains the results of an analysis of software errors by type. Since error data by category type was not available for Project 3, only error data from Projects 1 and 2 were analyzed.

Using the error category list in Table 2, the question naturally arises "how many of each type were there?" This analysis took place only at the major category level, and only errors which required a change to the source code or data base were examined. Categories "DATA INPUT ERRORS" and "DATA OUTPUT ERRORS" were combined into "INPUT/OUTPUT ERRORS". Categories "DATA HANDLING ERRORS" and "ARRAY PROCESSING ERRORS" were combined into "DATA HANDLING ERRORS. Categories "OPERATION ERRORS", "DOCUMENTATION ERRORS" and "PROGRAM EXECUTION ERRORS" are included in category "OTHER".

Figure 10 shows a percentage breakdown by major category for Projects 1 and 2. It also shows a percentage breakdown when Projects 1 and 2 are combined. Percentage breakdowns appear to be reasonable for the type software for each project. Project 2 is an on-line system and Project 1 is a batch system. Variations exist in the percentage breakdowns because of the kinds of operations the programs are performing. Project 1 programs are performing mostly data

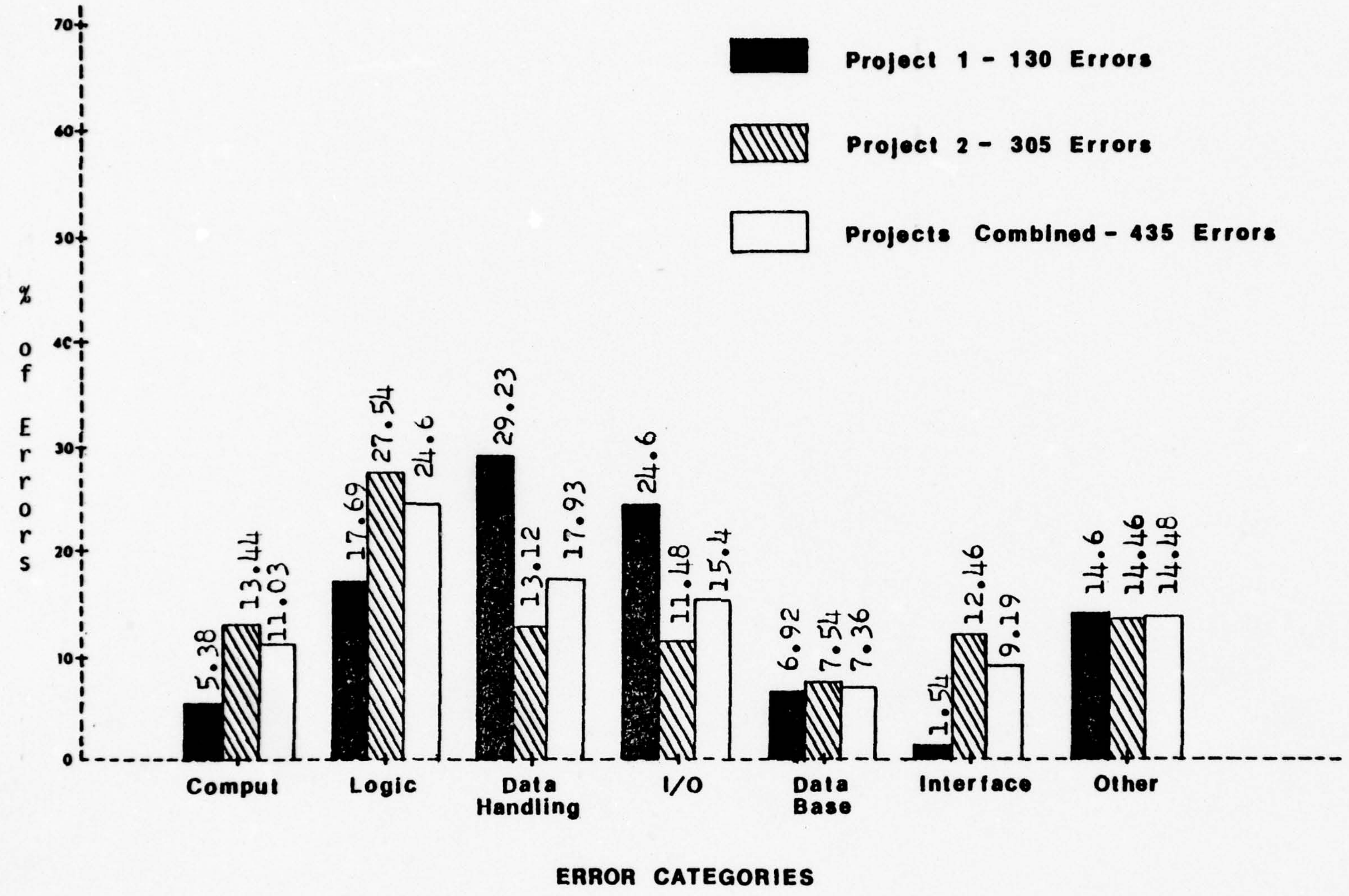


Figure 10. Percentage of Error Types for Projects 1 and 2

input/output and data transfers. The logic within the programs is not very complicated. On the other hand the programs in Project 2 are doing more computations on the data and the logic flow is more complicated.

The data for each project supports the notion that the distributions of the types are application dependent. However, when the data from the two projects are combined, the relative importance of the percentages changes. As one can see the logic type has the highest percentage (24.6% vs 17.9%). It is suggested that this will be true in the general case.

Summary of Regression Analysis

Regression Analysis Concepts

Situations exist where there are two or more variables that are functionally related. The problem is to understand this relationship. This is not an easy thing to do since the relationship may be a simple or a complex one. Most often, the functional relationship is extremely complex, or completely unknown. The goal is to obtain a better understanding of the relationship and then use that information for prediction, process optimization or control.

The technique usually employed in these investigations of relationships between variables is known as regression analysis. Regression analysis assumes the existence of a functional relationship

$$Y = F[x(1), x(2), \ldots, x(n); B(0), B(1), \ldots, B(n)] + e,$$

where Y is the response (or dependent) variable, $x(i)$ $(i=1,2,\ldots,n)$ are the independent variables, $B(j)$ $(j=0,1,\ldots,n)$ are unknown parameters, and e is a random error component. The problem consists of estimating the unknown parameters in the above equation.

The kind of relationship between Y and the independent variables determine the type of regression analysis to perform. If the assumption of linearity appears reasonable then linear regression analysis can be used. If linearity does not seem reasonable then some other analysis technique must be employed. Since this research assumes linearity, linear regression analysis techniques are used. The rest of this section briefly discusses linear regression analysis as related to this paper.

If one wishes to determine the relationship between a single independent variable, say x, and a single response or dependent variable, say Y, then use simple linear regression techniques. The equation to predict would be in the form

$$Y = B(0) + B(1)x + e,$$

where "B(0)" is the intercept, "B(1)" is the slope and "e" is the random error component. The procedure is to use the method of least squares to estimate "B(0)" and "B(1)" [68].

If one wishes to determine the relationship between many independent variables and a single dependent variable then use multiple linear regression techniques. The equation to predict would be in the form

$$Y = B(0) + B(1)x(1) + B(2)x(2) + \ldots + B(n)x(n) + e,$$

where B(i)'s are the unknown parameters (regression coefficients) to be estimated using the method of least squares . Y is the dependent variable, and x(i)'s are the known independent variables.

For our purpose, the independent variables are the characteristic metrics (see Table 4). The equation would be, if all metrics were included, similar to the following equation

$$N = B(0) + B(1)LC + B(2)IO + B(3)CO + B(4)DH + B(5)PC + B(6)UBR + B(7)EXIT + B(8)STOP + B(9)OSC + B(10)CC + B(11)TS + B(12)NEX + B(13)NNEX + B(14)FD + B(15)RD + B(16)DD + B(17)TD + B(18)DR + B(19)NCO + B(20)PAR + B(21)NL + B(22)RW,$$

where B(i)'s are estimated using project data from Tables 5-7.

Most organizations do not want to expend the resources to collect data. As the number of variables increase so does the collection and maintenance costs for metric and error data. Therefore, it is desirable to have a minimum number of metric variables to collect and maintain. Thus, the objectives of the regression analysis are to determine which metric variables singly correlated with the number of errors and to determine the "best" groups with the minimum number of variables to predict the number of errors in a program.

The regression results presented in the following sections are summaries of outputs from the Statistical Analysis System 76 [69]. For the interested reader reference [69] explains different techniques for determining which variables of a collection of independent variables should most likely be included in a regression model. Since all techniques (forward, backward, stepwise, maximum r-square, and minimum r-square) were available in SAS76, they were all used initially to screen the independent variable list. Two statistics, "r-square" and "F", are used to determine if a linear relationship exists. "R-square" is the proportion of variability in N that is explained by the relationship between N and the independent variables (characteristics metrics). "F" is a well known statistic used in tests of hypothesis.

Simple Linear Regression Analysis Results

Each variable represents a metric which serves to measure to some degree the number of program errors. In order to correlate the metric with the number of errors (N), a single variable linear regression analysis was performed on each metric variable.

A test of hypothesis was conducted on each metric to determine if its regression on N was significant. All tests were set up in the following manner with a .05 level of significance

$$H(0): \quad B(1) = 0,$$

$$H(1): \quad B(1) \neq 0.$$

and under the assumption that the e(i)'s are normally distributed. The regression statistics for each respective project is summarized in Tables 11-13. The "Decision" column indicates if H(0) is rejected or accepted. Rejected

TABLE 11. SUMMARY OF SINGLE VARIABLE REGRESSION DATA FOR PROJECT ONE

Metric Variable	R-square	F Value	Pr > F	Decision
LC	0.944435	50.99	0.0057	Reject H(0)
IO	0.212239	0.81	0.4349	Accept H(0)
CO	0.627788	5.06	0.1100	Accept H(0)
DH	0.943303	49.91	0.0058	Reject H(0)
PC	0.566996	3.93	0.1418	Accept H(0)
UBR	0.995852	720.16	0.0001	Reject H(0)
EXIT	0.285568	1.20	0.3535	Accept H(0)
STOP	0.056103	0.18	0.7013	Accept H(0)
OSC	0.074677	0.24	0.6564	Accept H(0)
CC	0.000000	0.00	1.0000	Accept H(0)
TS	0.677116	6.29	0.0871	Accept H(0)
NEX	0.945817	52.37	0.0054	Accept H(0)
NNEX	0.232533	0.91	0.4107	Accept H(0)
FD	0.111622	0.38	0.5827	Accept H(0)
RD	0.745010	8.77	0.0595	Accept H(0)
DD	0.430402	2.27	0.2292	Accept H(0)
TD	0.470261	2.66	0.2012	Accept H(0)
DR	0.885232	23.14	0.0171	Reject H(0)
NCO	0.095886	0.32	0.6121	Accept H(0)
PAR	0.421938	2.19	0.2355	Accept H(0)
NL	0.557331	3.78	0.1472	Accept H(0)
RW	0.877502	21.49	0.0189	Reject H(0)

TABLE 12. SUMMARY OF SINGLE VARIABLE REGRESSION DATA FOR PROJECT TWO

Metric Variable	R-square	F Value	Pr > F	Decision
LC	0.952913	242.85	0.0001	Reject H(0)
IO	0.769078	39.97	0.0001	Reject H(0)
CO	0.413572	8.46	0.0131	Reject H(0)
DH	0.585479	16.95	0.0014	Reject H(0)
PC	0.683309	25.89	0.0003	Reject H(0)
UBR	0.684094	25.99	0.0003	Reject H(0)
EXIT	0.521244	13.06	0.0035	Reject H(0)
STOP	0.281546	4.70	0.0509	Accept H(0)
OSC	0.000000	0.00	1.0000	Accept H(0)
CC	0.237354	3.73	0.0772	Accept H(0)
TS	0.579287	16.52	0.0016	Reject H(0)
NEX	0.869163	79.72	0.0001	Reject H(0)
NNEX	0.256202	4.13	0.0648	Accept H(0)
FD	0.001341	0.02	0.9011	Accept H(0)
RD	0.124145	1.70	0.2166	Accept H(0)
DD	0.267050	4.37	0.0585	Accept H(0)
TD	0.256989	4.15	0.0643	Accept H(0)
DR	0.884833	92.20	0.0001	Reject H(0)
PAR	0.807197	50.24	0.0001	Reject H(0)
NL	0.715260	30.14	0.0001	Reject H(0)
RW	0.594347	17.58	0.0012	Reject H(0)
NCO	0.709362	29.29	0.0002	Reject H(0)

TABLE 13. SUMMARY OF SINGLE VARIABLE REGRESSION DATA FOR PROJECT THREE

Metric Variable	R-square	F Value	Pr > F	Decision
LC	0.830952	216.28	0.0001	Reject H(0)
IO	0.081709	3.92	0.0541	Accept H(0)
CO	0.696377	100.92	0.0001	Reject H(0)
DH	0.751509	133.07	0.0001	Reject H(0)
PC	0.295987	18.50	0.0001	Reject H(0)
UBR	0.821589	202.62	0.0001	Reject H(0)
EXIT	0.209254	11.64	0.0014	Reject H(0)
STOP	0.014800	0.66	0.4206	Accept H(0)
OSC	0.174450	9.30	0.0039	Reject H(0)
CC	0.001323	0.06	0.8103	Accept H(0)
TS	0.755609	136.04	0.0001	Reject H(0)
NEX	0.809248	186.67	0.0001	Reject H(0)
NNEX	0.271785	16.42	0.0002	Reject H(0)
FD	0.010272	0.46	0.5027	Accept H(0)
RD	0.162182	8.52	0.0055	Accept H(0)
DD	0.291975	18.14	0.0001	Reject H(0)
TD	0.286922	17.70	0.0001	Reject H(0)
DR	0.799424	175.37	0.0001	Reject H(0)
NCO	0.023480	1.06	0.3093	Accept H(0)
PAR	0.647323	80.76	0.0001	Reject H(0)
NL	0.668134	88.58	0.0001	Reject H(0)
RW	0.742020	126.56	0.0001	Reject H(0)

means that the regression of the metric on N is significant. Accepted means that the regression is not significant.

A summary of the significance of the regression of single metrics on N is presented in Table 14. An "*" means the regression is significant. It is significant that "LC" is the highest(regardless of software type, operating mode or other project differences) in Projects 2 and 3 and second in Project 1. Variables "DH" and "RW" were also significant for all projects.

Although most variables turned out to be significant for one or the other projects, "LC" and "UBR" are more important since these can be obtained before coding starts (they can be read directly from "control flow charts). The predicted linear equations for all variables by project is contained in Table 15. An "*" means no equation was predicted for this project.

Multiple Linear Regression Analysis Results

The purpose of the multiple regression analysis was to determine which metric variables should most likely be included in a regression model. Mainly we were interested in screening the list of 22 metric variables shown in Table 4 to eliminate the ones that did not influence software error data.

TABLE 14. SUMMARY OF SIGNIFICANCE OF REGRESSION FOR ALL PROJECTS

Metric Variable	Project		
	One	Two	Three
LC	*	*	*
DH	*	*	*
UBR	*	*	*
DR	*	*	*
RW	*	*	*
CO		*	*
PC		*	*
EXIT		*	*
NEX		*	*
TS		*	*
PAR		*	*
NL		*	*
NCO		*	
OSC			*
NNEX			*
DD			*
TD			*
IO			
STOP			
CC			
FD			
RD			

TABLE 15. SINGLE VARIABLE MODELS FOR PREDICTING THE NUMBER OF ERRORS IN COBOL PROGRAMS

Metric Variable	Project One		Project Two		Project Three	
	Intercept	Slope	Intercept	Slope	Intercept	Slope
LC	2.902	0.587	3.7311	0.319	4.355	0.038
IO	*	*	5.848	0.303	*	*
CO	*	*	9.319	0.362	3.157	0.228
DH	4.623	0.404	5.640	0.104	1.525	0.040
PC	*	*	8.567	0.371	8.758	0.041
UBR	-1.683	1.047	12.621	0.122	4.110	0.060
EXIT	*	*	9.222	0.714	7.753	0.399
STOP	*	*	*	*	6.492	0.234
CC	*	*	*	*	*	*
TS	*	*	2.645	0.022	0.632	0.010
NEX	*	*	- 0.422	0.056	2.272	0.012
NNEX	*	*	*	*	3.579	0.020
FD	*	*	*	*	*	*
RD	*	*	*	*	*	*
DD	*	*	*	*	3.3839	0.024
TD	*	*	*	*	3.625	0.022
DR	-1.153	0.064	- 0.973	0.010	2.325	0.005
NCO	*	*	- 0.153	0.088	*	*
PAR	*	*	7.294	0.334	2.871	0.106
NL	*	*	- 0.326	0.017	- 0.477	0.010
RW	2.661	0.139	6.015	0.043	1.127	0.014

Analysis Using all Metric Variables

The basic approach to this part of the analysis was to study the outputs from all five techniques referenced above and select the sets of variables most frequently included for each project. Table 16 lists a few of the sets of variables selected for all projects. The highest "r-square" value is given when the same set was selected more than once. Column "Decision" reflects the decision relative to the hypothesis

$$H(0): \quad B(1)=B(2)\ldots=B(n) = 0,$$

against

$$H(1): \quad B(i)\neq 0 \text{ for at least one } i.$$

and under the assumption that the e(i)'s are normally distributed. At a .05 significance level, at least one B(i) is significantly different from zero for all selected groups. Table 17 contains the final list of variables chosen from those shown in Table 16. Table 18 contains a summary of the predicted equations for each set of variables by project.

The preceding discussion was based upon an objective analysis of 22 variables without considering their relationship to each other. Since some of the 22 variables were known to be functionally related, i.e., TS = NEX + NNEX and TD = FD + RD + DD, a subset of variables considered unrelated was selected and used in a regression analysis. These variables are LC, IO, CO, DH, PC, UBR, EXIT, STOP, OSC, CC, TD, PAR, and DR. The results from this analysis is presented in the next section.

TABLE 16. VARIABLES SELECTED FOR MODELS

No in Group	Variables Selected	R Square	F Value	Pr>F	Decision
2	LC UBR	0.9585	127.11	0.0001	R H(0)
	LC STOP	0.9911	612.28	0.0001	R H(0)
	UBR EXIT	0.9988	855.77	0.0012	R H(0)
3	UBR EXIT NNEX	0.9999	999.99	0.0001	R H(0)
	LC UBR CC	0.9935	850.6C	0.0001	R H(0)
	LC CO STOP	0.9999	611.51	0.0001	R H(0)
	LC CC NEX	0.9946	506.4C	0.0001	R H(0)
4	UBR IO EXIT NNEX	1.0000	999.99	0.0001	R H(0)
	DD TD DR RW	1.0000	999.99	0.0001	R H(0)
	LC CO UBR STOP	0.9915	461.51	0.0001	R H(0)
	LC EXIT CC NEX	0.9949	441.86	0.0001	R H(0)
	LC UBR OSC CC	0.8660	66.20	0.0001	R H(0)
	PC STOP CC NEX	0.8784	74.07	0.0001	R H(0)
5	UBR EXIT NNEX TS RW	1.0000	649.7C	0.0001	R H(0)
	LC CO UBR STOP DR	0.9955	354.02	0.0001	R H(0)
	LC CO UBR DR PAR	0.9955	354.02	0.0001	R H(0)
	LC PC EXIT STOP NEX	0.9975	652.77	0.0001	R H(0)
	LC UBR OSC CC FD	0.8716	54.29	0.0001	R H(0)
	PC STOP CC NNEX NL	0.8808	59.08	0.0001	R H(0)
10	LC IO CO UBR EXIT STOP DD TD DR PAR	0.9995	643.00	0.0001	R H(0)
	LC IO CO DH PC UBR DD TD DR PAR	0.9951	645.00	0.0001	R H(0)
	LC IO CO UBR CC FD DR NL PAR RW	0.9927	411.44	0.0002	R H(0)
	LC PC STOP NEX FD RD DD DR TS RW	0.9999	2310.7	0.0001	R H(0)
	LC IO CO UBR STOP OSC CC FD RD TD	0.8662	27.25	0.0001	R H(0)

BEST AVAILABLE COPY

TABLE 16. VARIABLES SELECTED FOR MODELS--Continued

No in Group	Variables Selected	R Square	F Value	Pr>F	Decision
10	LC PC CC NEX NNEX RD DD TD NL TS	0.9135	36.95	0.0001	R H(0)
	LC IO CO DH PC UBR EXIT OSC CC TD	0.9955	408.15	0.0001	R H(0)
	LC IO CO UBR STOP OSC CC NEX FD DR	0.8833	27.84	0.0001	R H(0)
	DH PC STOP CC NEX NNEX FD RD TS NL	0.8908	28.55	0.0001	R H(0)
15	LC IO CO UBR EXIT CC FD RD STOP TD DR NL PAR RW DD	1.0000	99999.	0.0001	R H(0)
	LC IO DH PC UBR CC NEX RD NNEX DD TD DR NCO TS NL	0.9216	23.49	0.0001	R H(0)
	LC CO DH PC UBR OSC CC FD RD DD TD NCO NL TS DR	0.9186	22.56	0.0001	R H(0)
20	LC IO CO DH PC UBR STOP CC OSC TS NEX NNEX FD RD DD TD DR NCO NL PAR	0.9244	15.28	0.0001	R H(0)
	LC IO CO DH PC UBR EXIT CC STOP OSC TS NEX NNEX FD RD DD TD DR NL PAR	0.9244	13.98	0.0001	R H(0)

TABLE 17. FINAL LISTS OF METRIC VARIABLES SELECTED

Metric Variable	Number of Variables in models				
	2	5	10	15	20
LC	*	*	*	*	*
UBR	*	*	*	*	*
IO		*	*	*	*
DH		*	*	*	*
EXIT		*	*	*	*
CO			*	*	*
PC			*	*	*
OSC			*	*	*
CC			*	*	*
STOP				*	*
TD			*	*	*
DR				*	*
PAR				*	*
NL				*	*
RW				*	*
TS					*
NNEX					*
FD					*
DD					*
NCO					*
NEX					
RD					

TABLE 18. MULTIPLE VARIABLE MODELS FOR PREDICTING THE NUMBER OF ERRORS IN COBOL PROGRAMS

PROJECT NO.	MODELS *
1	$\hat{N} = -1.556 + 0.029\text{LC} + 0.998\text{UBR}$
2	$\hat{N} = 3.971 + 0.302\text{LC} + 0.009\text{UBR}$
	$\hat{N} = -1.506 + 0.256\text{LC} + 0.124\text{IØ} - 0.003\text{DH} + 0.003\text{UBR} - 0.130\text{EXIT}$
3	$\hat{N} = 4.072 + 0.022\text{LC} + 0.027\text{UBR}$
	$\hat{N} = 3.327 + 0.018\text{LC} + 0.006\text{IØ} + 0.007\text{DH} + 0.0246\text{UBR} - 0.048\text{EXIT}$
	$\hat{N} = 5.633 + 0.020\text{LC} + 0.019\text{IØ} - 0.015\text{CØ} + 0.012\text{DH} - 0.009\text{PC} + 0.022\text{UBR} - 0.029\text{EXIT} + 0.049\text{ØSC} - 2.392\text{CC} - 0.001\text{TD}$
	$\hat{N} = 7.062 + 0.014\text{LC} + 0.032\text{IØ} + 0.002\text{CØ} + 0.015\text{DH} + 0.001\text{PC} + 0.034\text{UBR} - 0.012\text{EXIT} - 1.927\text{STØP} + 0.055\text{ØSC} - 2.087\text{CC} + 0.003\text{TD} + 001\text{DR} - 0.029\text{PAR} - 0.002\text{NL} - 0.001\text{RW}$
	$\hat{N} = 5.922 - 0.323\text{LC} - 0.302\text{IØ} - 0.312\text{CØ} - 0.299\text{DH} - 0.342\text{PC} - 0.275\text{UBR} - 0.047\text{EXIT} - 1.449\text{STØP} - 0.271\text{ØSC} - 2.589\text{CC} + 0.324\text{TS} - 0.336\text{NNEX} + 0.065\text{FD} - 0.078\text{DD} + 0.094\text{TD} + 0.002\text{DR} + 0.012\text{NCØ} - 0.044\text{PAR} - 0 - 011\text{NL} + 0.001\text{RW}$

*O's are slashed to distinguish from zeros.

Analysis of Reduced Sets of Unrelated Variables

It is appropriate at this time to point out that analysis was not performed for a project containing a number of observations less than the number of metric variables.

All tests of hypothesis were set up in the following manner with a 0.05 level of significance

$$H(0): \quad B(1)=B(2)=\ldots=B(n)=0$$

against

$$H(1): \quad B(i) \neq 0, \text{ for at least one } i,$$

and under the assumption that the e(i)'s are normally distributed. The results of this analysis are summarized in Table 19. The regression is significant. Therefore, this final set of 13 metrics is a good predictor of software errors.

Conclusions

The main purpose of the preceding analysis was to determine if program characteristics metrics, which measure program complexity, are predictors of the number of errors in COBOL programs. It was shown, through simple linear and multiple linear regression analysis, that the number of errors in a COBOL program is a function of its structure which is measured by characteristics metrics.

A set of 13 unrelated metrics was chosen as the final group of metrics to predict the number of software errors. The next chapter will specifically discuss how this set is used as a measure of program complexity.

TABLE 19. SUMMARY OF REGRESSION ANALYSIS OF REDUCED SET OF VARIABLES

PROJECT	MODELS *	R SQUARE	F VALUE	PR > F	DECISION
1	$\hat{N}$ = 1.180 + 1.088UBR - 0.460EXIT	0.9988	855.77	0.0012	R H (0)
2	$\hat{N}$ = 1.92 + 0.191LC + 0.108CØ + 0.112UBR - 0.260PAR + 0.003DR	0.9968	510.88	0.0001	R H (0)
	$\hat{N}$ = - 3.836 + 0.231LC + 0.031CØ + 0.033DH + 0.103PC + 0.042UBR - 0.202EXIT + 6.133STØP - 0.011TD - 0.229PAR + 0.004DR	0.9993	451.06	0.0002	R H (0)
3	$\hat{N}$ = 4.072 + 0.022LC + 0.027UBR	0.8463	118.40	0.0001	R H (0)
	$\hat{N}$ = 5.583 + 0.021LC + 0.017IØ + 0.026UBR + 0.055ØSC - 2.025CC	0.8702	53.61	0.0001	R H (0)
	$\hat{N}$ = 7.188 + 0.010LC + 0.026IØ + 0.010DH + 0.030UBR - 0.019EXIT - 2.105STØP + 0.051ØSC - 2.146CC - 0.037PAR + 0.001DR	0.8844	26.77	0.0001	R H (0)
	$\hat{N}$ = 7.215 + 0.010LC + 0.027IØ + 0.008CØ + 0.11DH + 0.002PC + 0.032UBR - 0.015EXIT - 2.104STØP + 0.052ØSC - 2.143CC - 0.0002TD - 0.043PAR + 0.001DR	0.8845	18.84	0.0001	R H (0)

*O's are slashed to distinguish them from zeros.

VI. A MEASURE OF PROGRAM COMPLEXITY

Introduction

Initially it was hypothesized that the number of software errors could be predicted from the internal complexity of the programs. But what does one mean by "internal complexity"? Is it a property that can be observed and measured, and perhaps even related to the number of errors in programs?

Complexity of any object is some measure of the mental effort [4,17] required to understand that object. In general usage, the complexity of an object is a function of the relationships among the components of the object. As applied to computer programs, it is a measure of the internal structural characteristics of the program. The previous chapter presented the results of an analysis that showed that actual program characteristics were related to the number of program errors. The purpose of this chapter is to define a "program complexity measure" from these basic characteristics.

Measures of Program Complexity

A program is made up of many components such as object instructions, data base descriptions, external data bases, other external programs such as the operating system and other application programs, program logic, etc. These components and the relationships among them determine program complexity. The problem is to measure the degree to which certain relationships exist within a program. For example, the number of input/output statements measures, in some degree, whether an input/output relation exists between the program and a data base. Complexity when applied to a specific relation is called local complexity. The complexity of a program as a whole is defined in terms of local complexities. There are 7 local complexities: control flow complexity, input/output complexity, data use complexity, computational complexity, data transfer complexity, structure design complexity, and interface complexity.

Control flow complexity is defined as the number of logical relationships present in the source code. In a COBOL program, these relations are manifested as IF, GOTO, STOP, and PERFORM...UNTIL statements, and AND and OR conditions. The Control Flow Complexity metric, CFC, can be numerically evaluated for each program by calculation:

$$CFC = LC + UBR + STOP$$

A Normalized Control Flow metric, NCFC, is defined as NCFC = CFC/1000.

Input/output complexity is defined as the number of I/O statements. The Input/Output Complexity metric, abbreviated as IOC, is

$$IOC = IO.$$

Data use complexity is defined as the ratio of data references (actual references to data items) to total data definitions. Many data reference errors are made because the assumptions made when defining data differs from the assumptions made when using the data. For example, when defining a data item as ALPHABETIC, it is assumed that only alphabetic data will be stored; but, when numeric data is stored in the data item it is assumed that the data item is declared "numeric" or "alphanumeric". As the number of data references increases relative to total definitions, the data use complexity increases. The Data Use Complexity metric, abbreviated as DUC, is calculated as

$$DUC = DR/TD.$$

Computational complexity is defined as the number of arithmetic statements such as ADD, MULTIPLY, COMPUTE, etc. The metric for this complexity is abbreviated as COC where

$COC = CO.$

Data transfer complexity is measured by the number of data transfer statements. This complexity metric, abbreviated as DHC, is evaluated by counting the number of data transfer statements ($DHC = DH$).

Interface complexity is measured by the number of system interfaces (number of system routines called) and the number of application routine interfaces (number of internal and external application routines called). The Interface Complexity metric, abbreviated as IC, is defined as follows:

$$IC = OSC + CC + PC,$$

AD-A046 873 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO F/G 9/2
A MODEL FOR ESTIMATING THE NUMBER OF RESIDUAL ERRORS IN COBOL P--ETC(U)
JUN 77 C E MARTIN
UNCLASSIFIED AFIT-CI-77-97 NL

2 OF 2
ADA046873

END
DATE
FILMED
12-77
DDC

where

OSC = number of system program interfaces

CC = number of compiler calls such as COPY

PC = number of application (internal and external) routine interfaces.

Structural design complexity is a measure of the number of distinct routines (number of components) in a program. The Structural Design Complexity metric referred to as SC, is numerically evaluated for each program by calculating:

$$SC = (PAR - EXIT) + 1.$$

The 1 accounts for the main control flow routine.

Total complexity is a function of the 7 local complexities discussed above. The Total Complexity metric, referred to as TC, is defined as follows:

$$TC = CFC + IOC + DUC + COC + DHC + IC + SC.$$

A Normalized Total Complexity metric, referred to as NTC, is defined as TC = TC/1000.

Regression Analysis of Complexity Metrics

Heretofore, data from 3 projects were used. But here, the complexity metrics are analyzed using the original 3 data bases plus 1 data base consisting of all 3 projects' data combined. The purpose for mixing the sources of data is to observe what happens when different types of programs, developed by different organizations, are mixed. Hereafter, the combined data base is referred to as Project 4. There are 65 observations (programs) in this data base. Tables

20-23 summarize the regression statistics for each respective project. The "Decision" column reflects the decision at the .05 significance level relative to the hypothesis

$$H(0): \quad B = 0,$$

against

$$H(1): \quad B \neq 0,$$

and under the assumption that the e(i)'s are normally distributed. Table 24 presents a summary of the regression siqificance, at the .05 level, of the complexity metrics by project.

The "best" complexity models, predicted by the Maximum R-square technique, are listed by project in Table 25. The models for total complexity and the normalized metrics, TC, NTC and NCFC, are listed in Table 26. Actual data points (plotted as '+') and the regression line for project 3 are shown in Figures 11-20.

Conclusions

The purpose of this chapter was to define and develop a "program complexity measure" from 13 unrelated characteristics metrics selected from the analysis presented in the previous chapter. Seven local complexities were defined and used to develop a measure of total program complexity. Metrics for each complexity were defined also. It was shown, through simple linear and multiple linear regression analysis, that the number of errors in COBOL programs is a function of these complexity metrics.

Linear models developed from these metrics can be used to predict the number of errors in COBOL programs. The "best" single variable model for predicting errors is the Control Flow Complexity metric model. The "best" multiple variable model for predicting errors is the one that contains all 7 local complexity metrics. Analysis of Project 4 showed that the latter model can be used when dealing with many types of programs that are developed by different organizations. However, it is suggested that each organization estimate the model parameters relative to error data from its development projects.

TABLE 20. SUMMARY OF THE COMPLEXITY METRICS REGRESSION DATA FOR PROJECT 1

Complexity Metric Variable	R-square	F Value	Pr > F	Decision
CFC	0.975512	119.51	0.0016	Reject H(0)
IOC	0.212239	0.81	0.4349	Accept H(0)
DUC	0.258828	1.05	0.3814	Accept H(0)
COC	0.627788	5.06	0.1100	Accept H(0)
DHC	0.943303	49.91	0.0058	Reject H(0)
IC	0.091318	0.30	0.6212	Accept H(0)
SC	0.459725	2.55	0.2084	Accept H(0)
TC	0.934449	42.77	0.0073	Reject H(0)
NTC	0.934449	42.77	0.0073	Reject H(0)
NCFC	0.975512	119.51	0.0016	Reject H(0)

TABLE 21. SUMMARY OF THE COMPLEXITY METRICS REGRESSION DATA FOR PROJECT 2

Complexity Metric Variable	R-square	F Value	Pr > F	Decision
CFC	0.824537	56.39	0.0001	Reject H(0)
IOC	0.769078	39.97	0.0001	Reject H(0)
DUC	0.061079	0.78	0.3943	Accept H(0)
COC	0.413575	8.46	0.0131	Reject H(0)
DHC	0.585479	16.95	0.0014	Reject H(0)
IC	0.727358	32.01	0.0001	Reject H(0)
SC	0.660955	23.39	0.0004	Reject H(0)
TC	0.903222	112.00	0.0001	Reject H(0)
NTC	0.903222	112.00	0.0001	Reject H(0)
NCFC	0.824537	56.39	0.0001	Reject H(0)

TABLE 22. SUMMARY OF THE COMPLEXITY METRICS REGRESSION DATA FOR PROJECT 3

Complexity Metric Variable	R-square	F Value	Pr > F	Decision
CFC	0.845919	241.56	0.0001	Reject H(0)
IOC	0.081709	3.92	0.0541	Accept H(0)
DUC	0.385898	27.65	0.0001	Reject H(0)
COC	0.696377	100.92	0.0001	Reject H(0)
DHC	0.751509	133.07	0.0001	Reject H(0)
IC	0.341395	22.81	0.0001	Reject H(0)
SC	0.686814	96.49	0.0001	Reject H(0)
TC	0.806581	183.48	0.0001	Reject H(0)
NTC	0.806581	183.48	0.0001	Reject H(0)
NCFC	0.845919	241.56	0.0001	Reject H(0)

TABLE 23. SUMMARY OF THE COMPLEXITY METRICS REGRESSION DATA FOR PROJECT 4

Complexity Metric Variable	R-square	F Value	Pr > F	Decision
CFC	0.300097	27.01	0.0001	Reject H(0)
IOC	0.172233	13.11	0.0006	Reject H(0)
DUC	0.151020	11.21	0.0014	Reject H(0)
COC	0.346999	33.48	0.0001	Reject H(0)
DHC	0.270823	23.40	0.0001	Reject H(0)
IC	0.106096	7.48	0.0081	Reject H(0)
SC	0.206157	16.36	0.0001	Reject H(0)
TC	0.304956	27.64	0.0001	Reject H(0)
NTC	0.304956	27.64	0.0001	Reject H(0)
NCFC	0.300097	27.01	0.0001	Reject H(0)

TABLE 24. REGRESSION SIGNIFICANCE OF COMPLEXITY METRICS VERSUS NUMBER OF ERRORS

Complexity Metric Variable	Project			
	1	2	3	4
CFC	*	*	*	*
DEC	*	*	*	*
TC	*	*	*	*
NTC	*	*	*	*
NCFC	*	*	*	*
COC		*	*	*
IC		*	*	*
SC		*	*	*
IOC		*		*
DUC			*	*

TABLE 25. COMPLEXITY MODELS FOR PREDICTING THE NUMBER OF ERRORS IN COBOL PROGRAMS

PROJECT	MODELS *	R-SQUARE
1	$\hat{N}$ = 0.287 + 0.382CFC	0.97551
	$\hat{N}$ = 0.205 + 0.326CFC + 0.25CØC	0.99775
	$\hat{N}$ = 3.207 + 0.326CFC + 0.279CØC - 0.163IC	0.99999
	$\hat{N}$ = 3.156 + 0.326CFC - 0.017IØC + 0.283CØC - 0.146IC	1.0000
2	$\hat{N}$ = 8.943 + 0.096CFC	0.82454
	$\hat{N}$ = 2.015 + 0.083CFC + 0.233CØC	0.98091
	$\hat{N}$ = 0.845 + 0.074CFC + 0.225CØC + 0.017DHC	0.98787
	$\hat{N}$ =-0.903 + 0.071CFC + 0.263DUC + 0.225CØC + 0.020DHC	0.98978
	$\hat{N}$ = 0.098 + 0.097CFC + 0.270DUC + 0.216CØC + 0.062IC - 0.133SC	0.99130
	$\hat{N}$ =-1.014 + 0.086CFC + 0.021IØC + 0.302DUC + 0.206CØC + 0.062IC - 0.103SC	0.99193
	$\hat{N}$ =-1.291 + 0.079CFC + 0.019IØC + 0.314DUC + 0.208CØC + 0.005DHC + 0.056IC - 0.074SC	0.99199

*O's are slashed to distinguish them from zeros.

BEST AVAILABLE COPY

TABLE 25--continued

PROJECT	MODELS *	R-SQUARE
3	$\hat{N}$ = 4.069 + 0.024CFC	0.84591
	$\hat{N}$ = 3.581 + 0.021CFC + 0.006DHC	0.84804
	$\hat{N}$ = 3.451 + 0.021CFC + 0.008DHC - 0.004IC - 0.004IC	0.84937
	$\hat{N}$ = 3.253 + 0.021CFC + 0.004IØC + 0.007DHC - 0.004IC	0.84965
	$\hat{N}$ = 3.244 + 0.022CFC + 0.006IØC - 0.017COC + 0.009DHC - 0.005IC	0.85012
	$\hat{N}$ = 3.102 + 0.022CFC + 0.006IØC + 0.045DUC - 0.018COC + 0.009DHC - 0.006IC	0.85020
	$\hat{N}$ = 3.094 + 0.022CFC + 0.008IØC + 0.044DUC - 0.017COC + 0.009DHC - 0.005IC - 0.005SC	0.85025
4	$\hat{N}$ = 6.789 + 0.254CØC	0.34700
	$\hat{N}$ = 3.890 + 0.061IØC + 0.218CØC	0.37532
	$\hat{N}$ = 3.784 + 0.011CFC + 0.066IØC + 0.131CØC	0.39897
	$\hat{N}$ = 4.509 + 0.025CFC + 0.082IØC + 0.109CØC - 0.079SC	0.41620
	$\hat{N}$ = 4.433 + 0.028CFC + 0.091IØC + 0.119CØC + 0.027IC - 0.139SC	0.42830
	$\hat{N}$ = 2.483 + 0.026CFC + 0.088IØC + 0.527DUC + 0.115CØC + 0.024IC - 0.136SC	0.43592
	$\hat{N}$ = 2.845 + 0.029CFC + 0.093IØC + 0.495DUC + 0.119CØC - 0.007DHC + 0.028IC - 0.138SC	0.43695

TABLE 26. TOTAL AND NORMALIZED COMPLEXITY MODELS BY PROJECT

PROJECT	MODELS
1	$\hat{N} = -3.728 + 0.153TC$
	$\hat{N} = -3.728 + 153.060NTC$
	$\hat{N} = 0.287 + 381.860NCFC$
2	$\hat{N} = 1.374 + 0.042TC$
	$\hat{N} = 1.374 + 41.977NTC$
	$\hat{N} = 8.943 + 95.763NCFC$
3	$\hat{N} = 2.220 + 0.011TC$
	$\hat{N} = 2.220 + 11.222NTC$
	$\hat{N} = 4.069 + 23.830NCFC$
4	$\hat{N} = 7.489 + 0.012TC$
	$\hat{N} = 7.489 + 11.666NTC$
	$\hat{N} = 9.521 + 24.370NCFC$

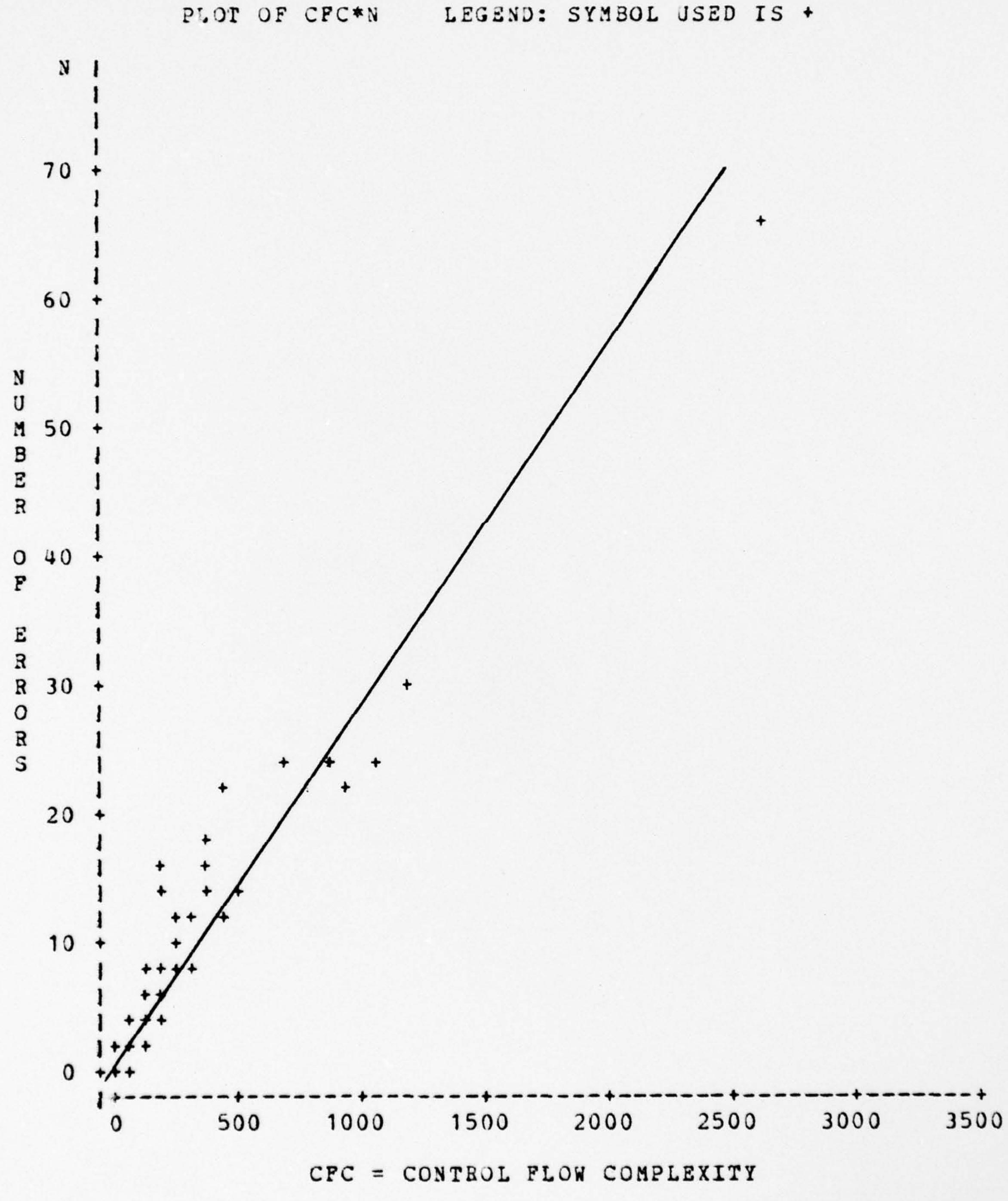


Figure 11. Plot of Control Flow Complexity Model for Project 3

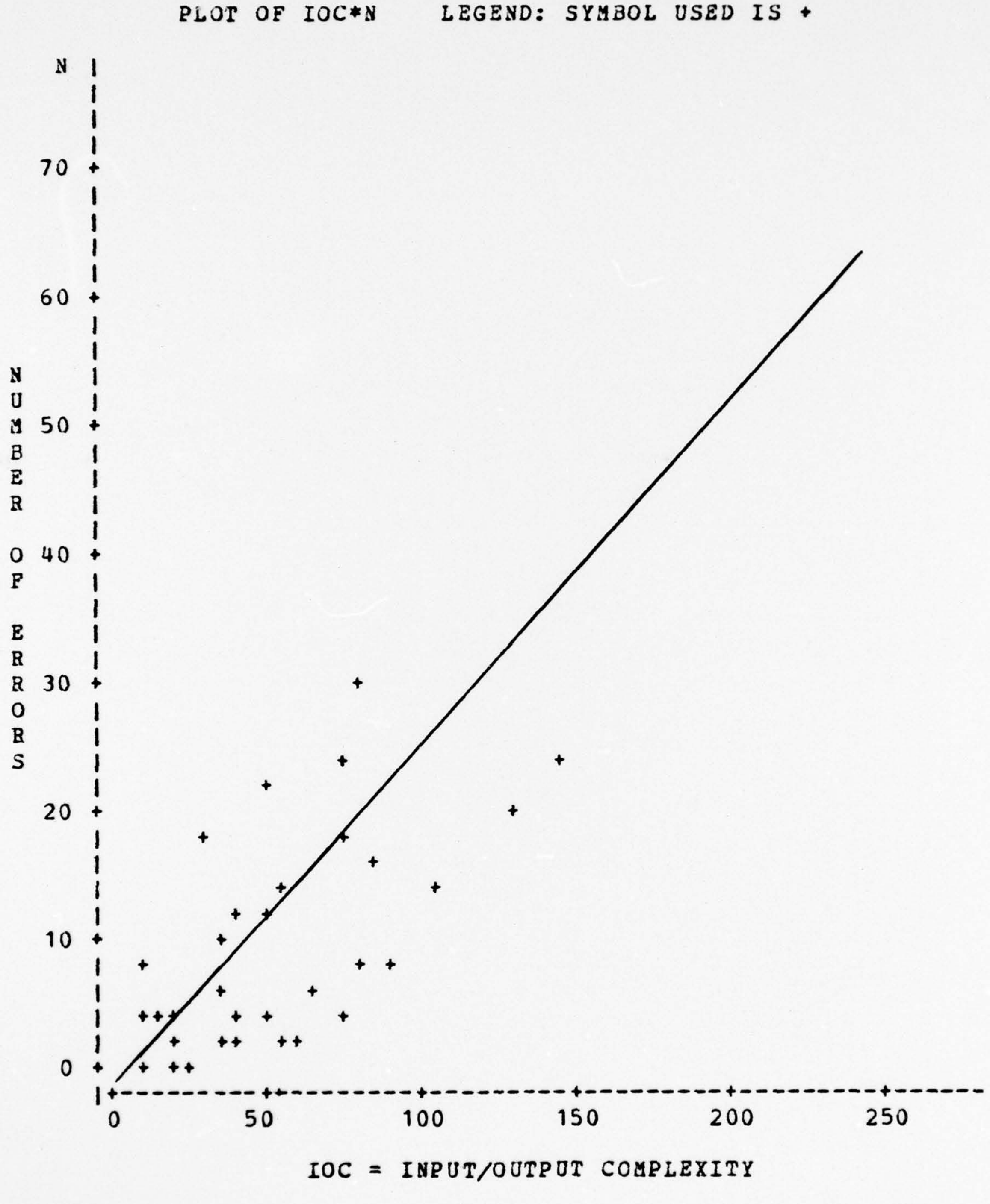


Figure 12. Plot of I/O Complexity Model for Project 3

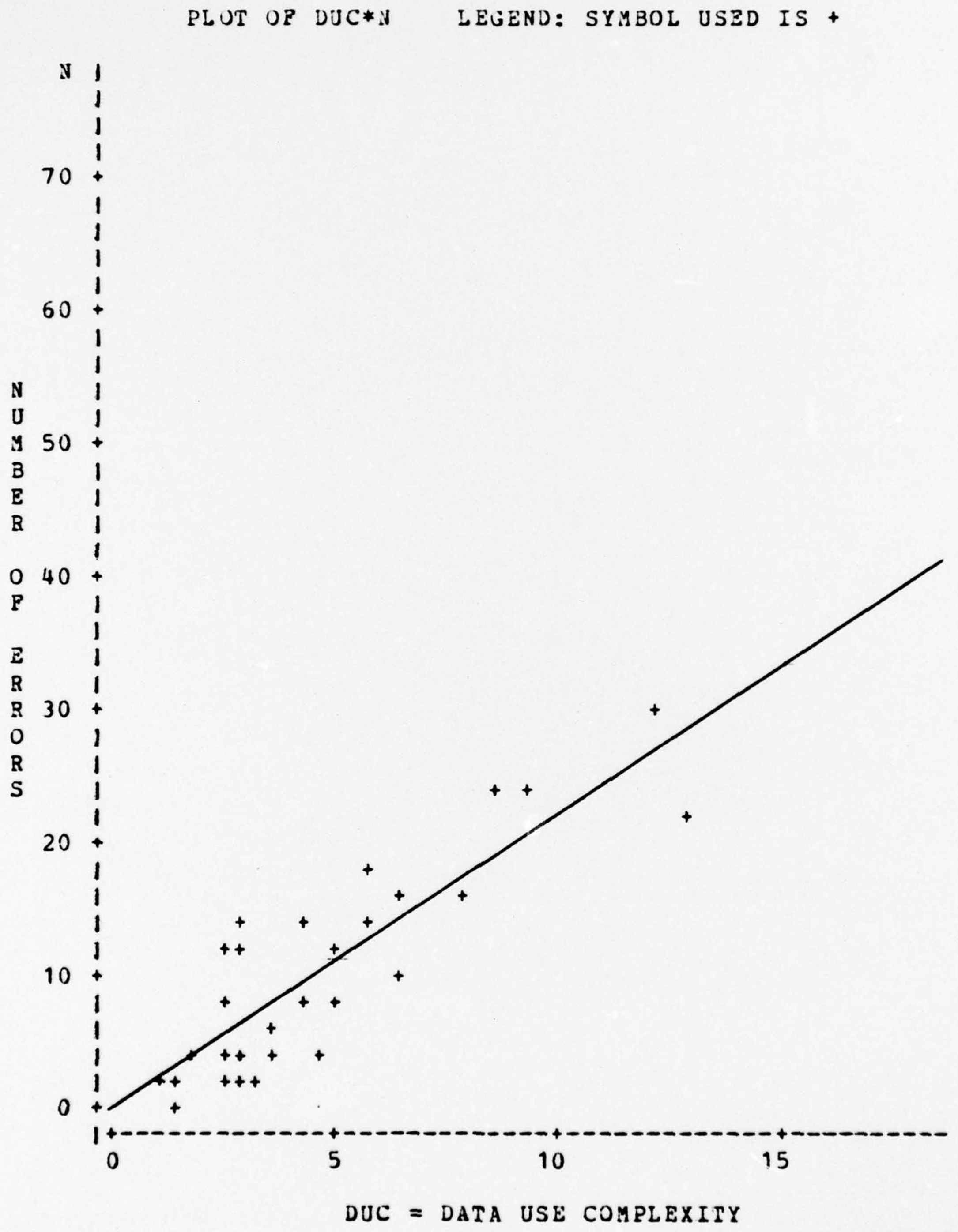


Figure 13. Plot of the Data Use Complexity Model for Project 3

PLOT OF COC*N LEGEND: SYMBOL USED IS +

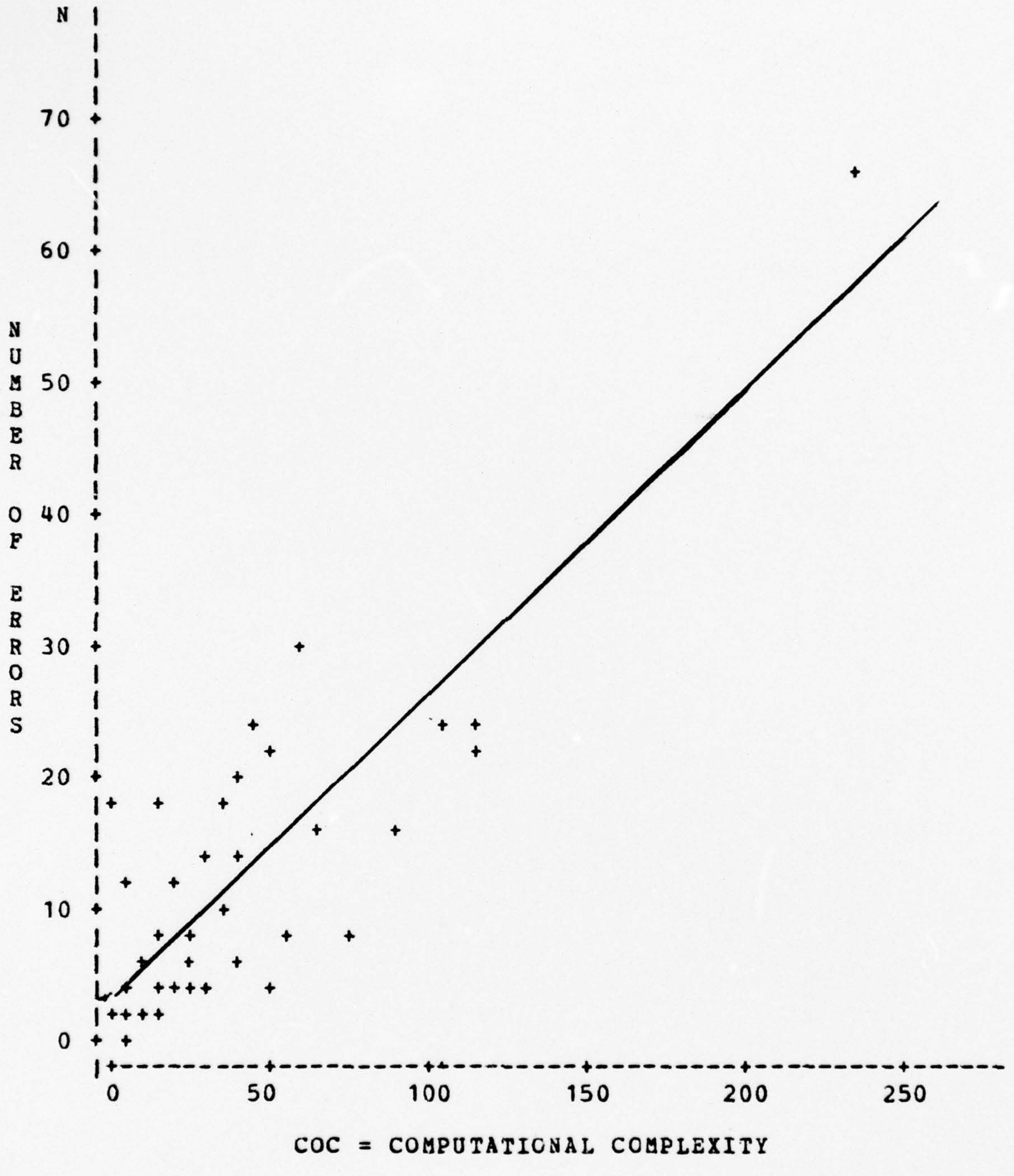


Figure 14. Plot of the Computational Complexity Model for Project 3

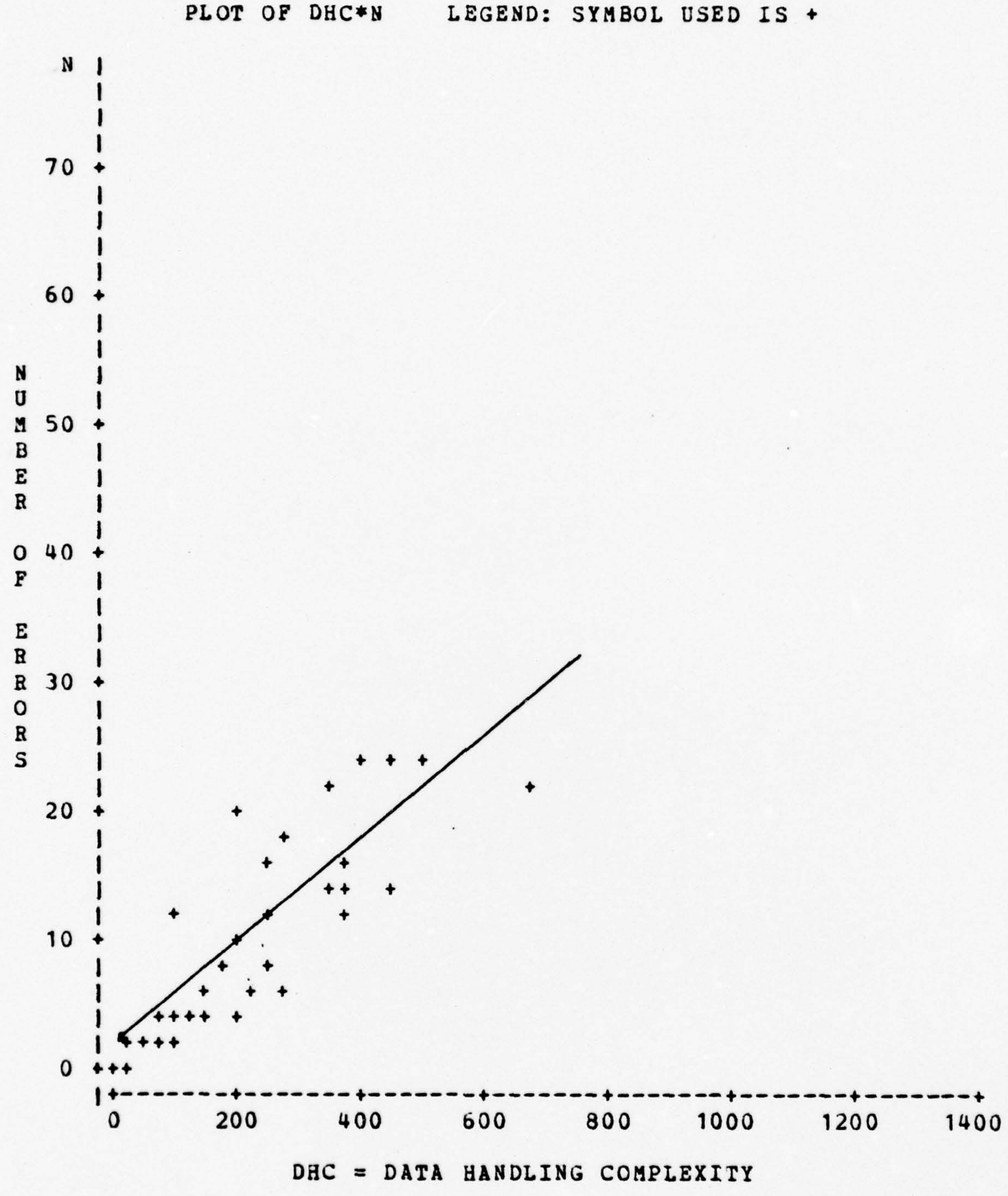


Figure 15. Plot of the Data Handling Complexity Model for Project 3

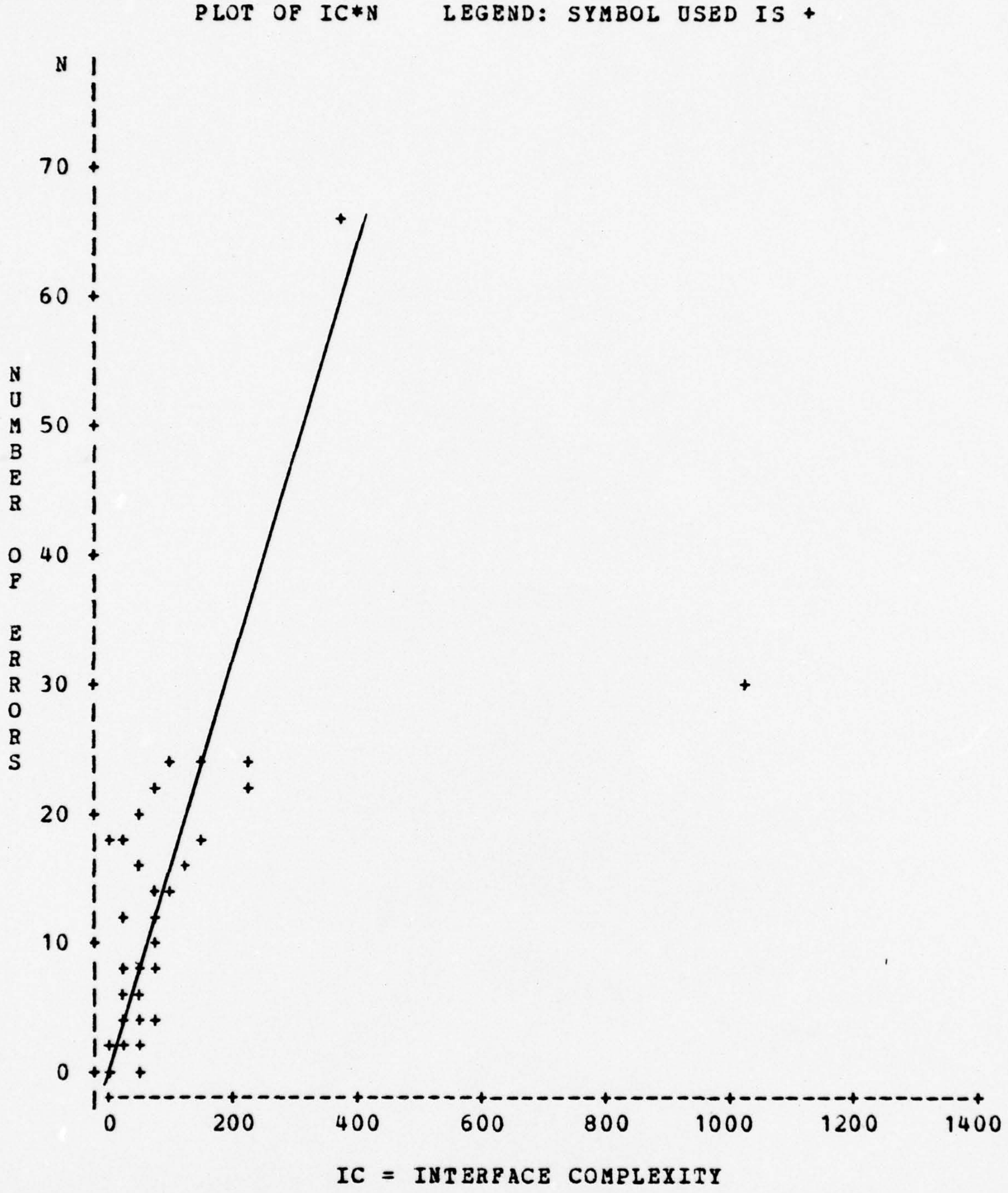


Figure 16. Plot of the Interface Complexity Model for Project 3

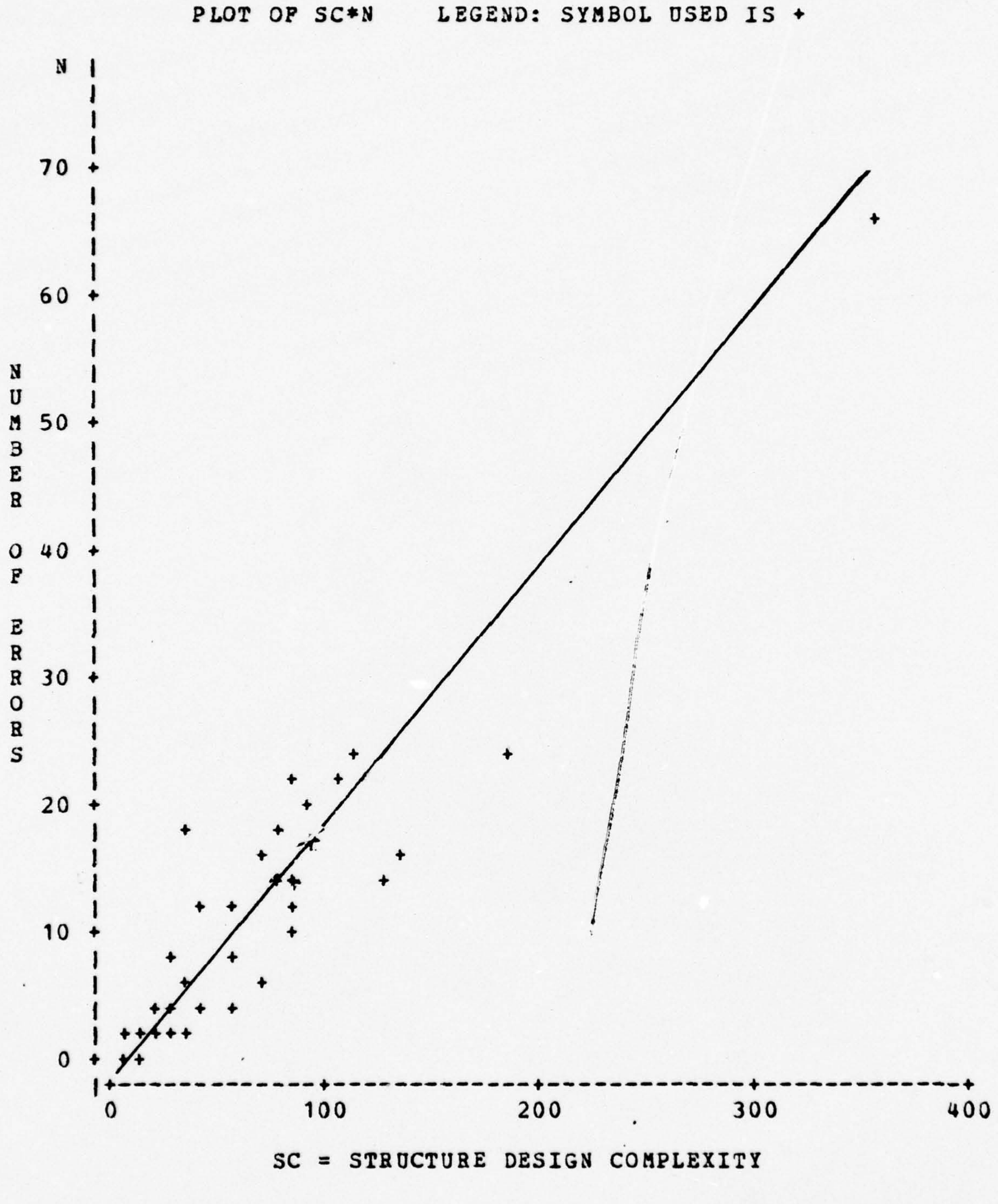


Figure 17. Plot of the Structure Design Complexity Model for Project 3

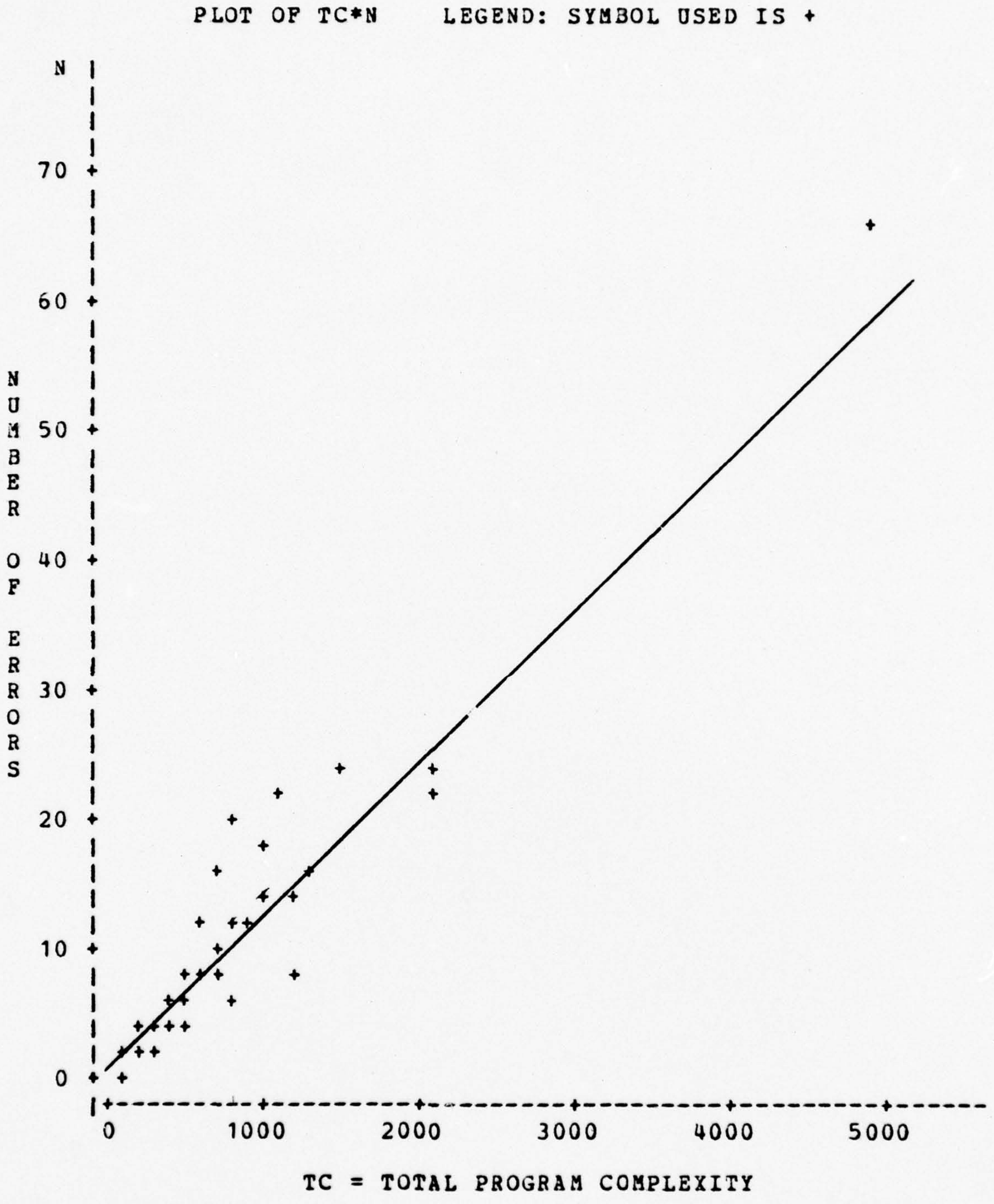


Figure 18. Plot of the Total Program Complexity Model for Project 3

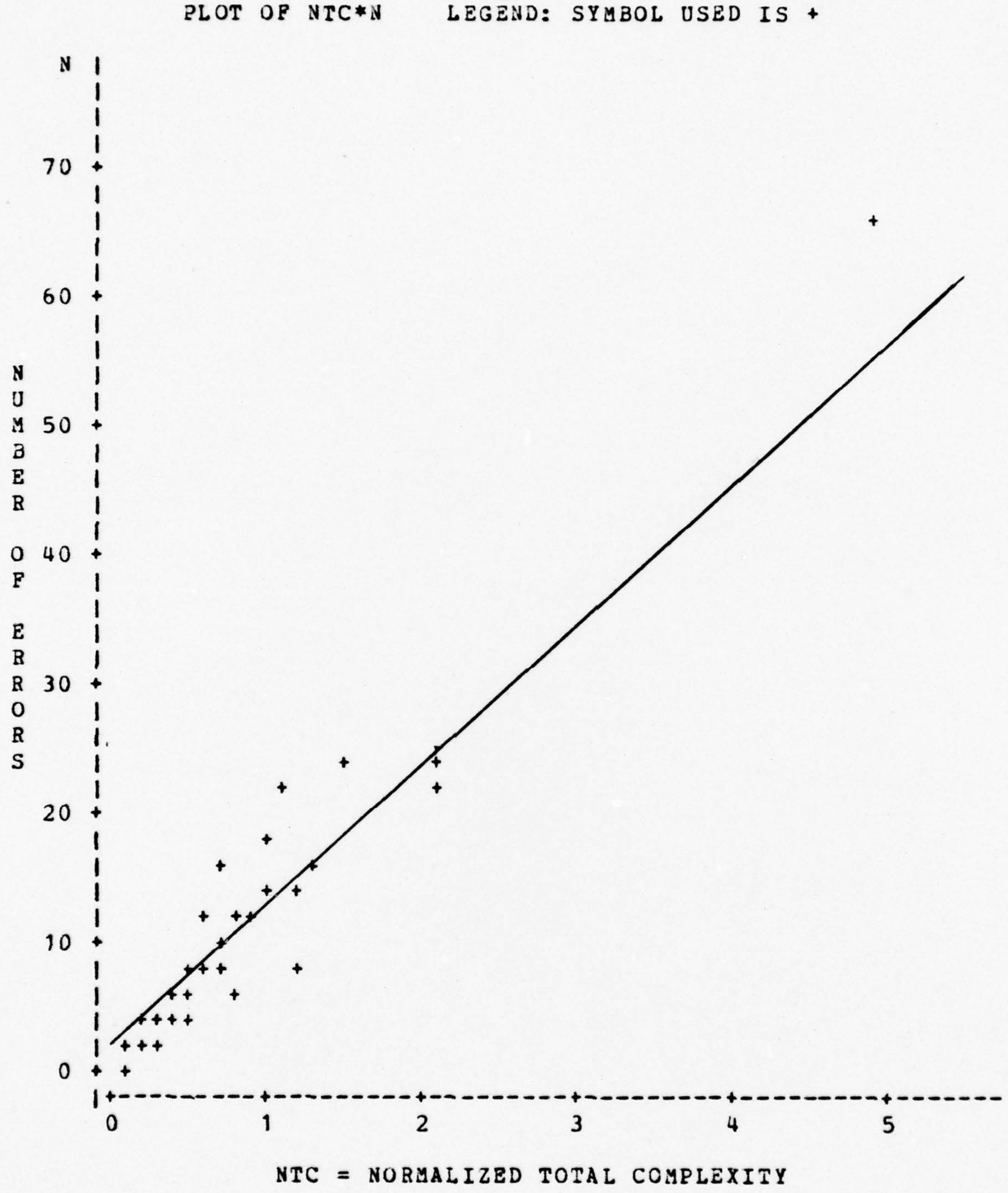


Figure 19. Plot of the Normalized Total Complexity Model for Project 3

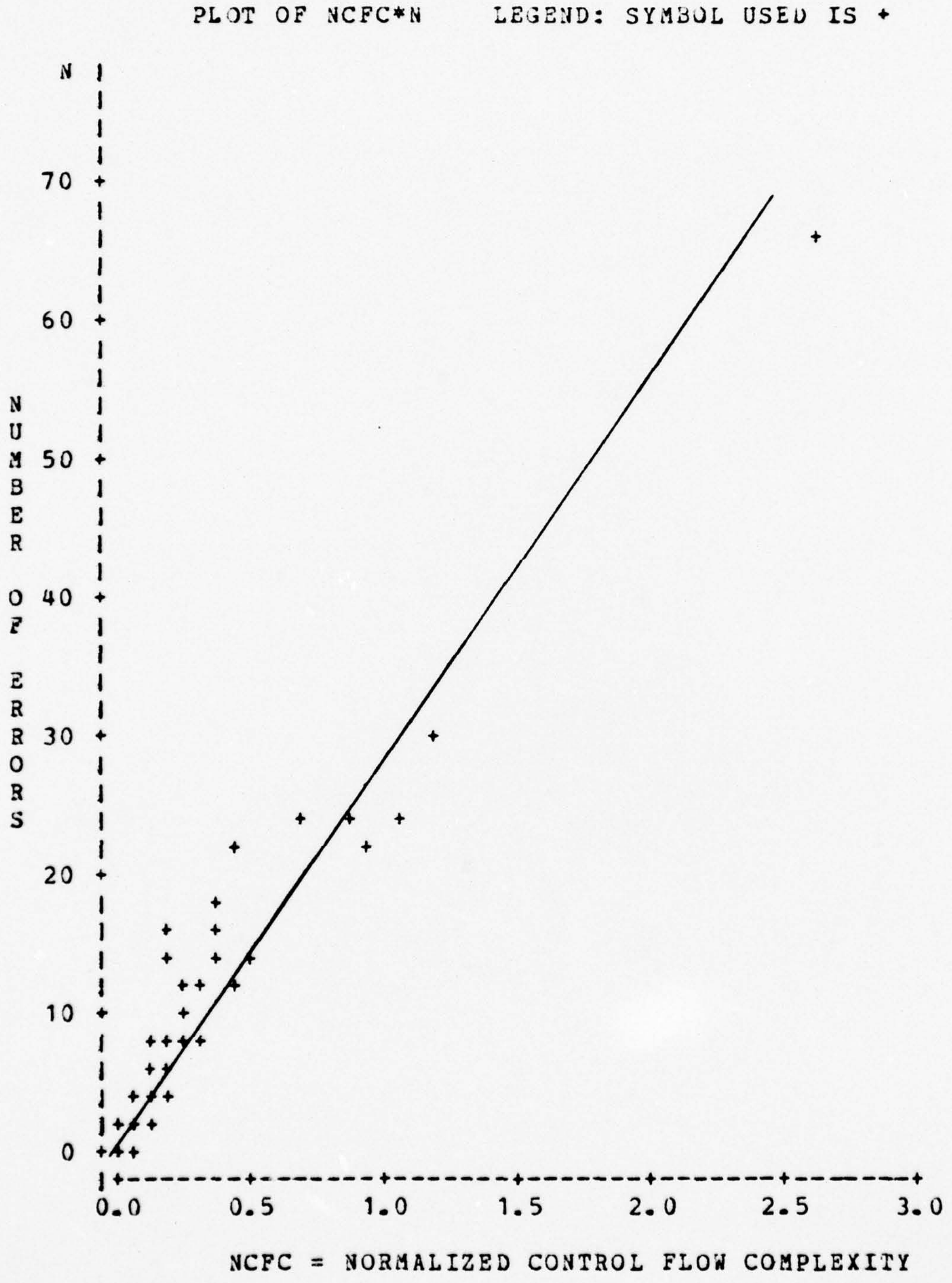


Figure 20. Plot of the Normalized Control Flow Complexity Model for Project 3

VII. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

Summary

The high incidence of errors in software is the underlying problem of software reliability. But, the most important unknown of software reliability is the number of residual errors in a program. If this number were available early in the software development stages, the software engineering process would be enhanced greatly. Several other unknowns could then be solved. One could determine when to stop testing a program, estimate the cost of maintenance and establish levels of confidence in programs and systems of programs, and develop more accurate software models that would model software failures more realistically. The results would be the ability to deal with all the unknowns of software reliability and reduce the overall cost of software.

The goals of this research were to determine if actual program characteristics are predictors of the number of errors in COBOL programs, to define program complexity measures from program characteristics, and to propose complexity models for predicting the number of errors in a COBOL program.

Through simple linear and multiple linear regression analysis of 3 sources of data, the number of errors in a COBOL program was shown to be a function of its structure which can be measured by 22 characteristics metrics. A set of 13 unrelated characteristics metrics was selected from the 22 characteristics metrics to define 7 local complexity metrics. The 7 local complexity metrics were predictors of the number of errors in a program. Consequently, these metrics were used to estimate models to predict errors. The "best" single variable model for predicting errors is the Control Flow Complexity metric model. The "best" multiple variable model for predicting errors is the one that contains all 7 local complexity metrics. Analysis of Project 4 showed that the latter model can be used when dealing with many types of programs that are developed by different organizations. However, each organization should estimate the model parameters relative to error data from its development projects. Results relative to Projects 1, 2, 3 and 4 are summarized in Appendices B, C, D and E respectively.

There are several applications for the complexity models. Some of them are:

1) Estimating the number of errors in programs,
2) Controlling the quality and structural complexity of programs during design [18],
3) Estimating and allocating resources for program maintenance [15,62],
4) Estimating a level of confidence in a program [4],

5) Developing failure, density, and reliability functions for software reliability, and

6) Establishing a cut-off point for debugging and testing computer software [10].

Regardless of application, however, it is necessary to estimate the number of errors. Once this number is available, the problems of software reliability can be treated more effectively. In order to illustrate the application of the complexity model in determining software reliability, Appendix F presents an example calculation for Project 2.

Conclusions

A detailed look at error types showed that logic and data handling were, percentagewise, the most frequent errors in Projects 2 and 1 respectively. However, when error data from both projects were combined, logic errors were the most frequent errors. It seems that the percentage of error types will vary depending on type of software; but, in general, logic errors will normally be the most frequent.

Twenty-two program characteristics metrics were analyzed by regression analysis techniques. Both single and multiple variable regression analysis showed that the relationship between the metrics and the number of errors was significant. Both single and groups of the structural characteristics metrics were good predictors of the number of errors. The number of logical conditions is the "best" single predictor. The number of unconditional branches is the "second best" single predictor. Different combinations of metrics were good predictors also. The metrics in the "best" equations, as determined by the Maximum R-square technique, varied by project. However, "LC" and "UBR" consistently appeared in the "best" equations. Thirteen unrelated metrics were selected to measure program complexity.

Total program complexity is measured by 7 local complexity metrics. Several complexity models are good predictors of the number of errors in COBOL programs. The "best" single variable model for predicting errors is the Control Flow Complexity metric model. The "best" multiple variable model for predicting errors is the one that contains all 7 local complexity metrics. The latter model can be used when dealing with many types of programs that are developed by different organizations.

Recommendations

One very worthwhile outcome of this study was a positive attitude toward being able to predict software errors from complexity measures. This paper only scratched the surface by showing that program complexity could be used to predict the number of errors in COBOL programs. However, the measures should apply to all languages. Other research areas are discussed below.

The results from the error type analysis indicate that error types did have some sort of distribution. Complexity metrics for specific error types would be very useful for costing and scheduling program maintenance.

The Data Use metric seems promising. It seems that the Reserve Word characteristic metric is related to this metric. More research is needed to determine if this is true.

It was shown that the number of errors are predictable from complexity measures. We hypothesize that the number of personnel assigned to a development project, total software cost, total development time, computer test time, maintenance cost, and program enhancement cost are also functions of program complexity measures. Additional research is needed to determine if this hypothesis is true.

REFERENCES

1. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19 - No. 5, May 1973, pp. 48-59.

2. T. W. Dolotta et al., Data Processing in 1980-85. New York: Wiley-Interscience, 1976.

3. "United States Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85)," Technology Trends: Software, Volume IV, October 1973.

4. Glenford J. Myers, Software Reliability Principles & Practices, A Wiley-Interscience Publication, John Wiley & Sons, New York.London.Sydney.Toronto, 1976.

5. B. W. Boehm, "Software Enginerring," IEEE Transactions on Computers, December 1976, pp. 1226-1241.

6. B. C. DeRose, "Managing the Development of Weapon System Software," in: Proceedings of Conference on Managing the Development of Weapon System Software, May 12, 1976, pp. 4-1 through 4-12.

7. W. L. Trainer, "Software: From Satan to Saviour," NAECON Proceeding, May 1973.

8. M. Lipow, "Maximum Likelihood Estimation of Parameters of a Software Time-To-Failure Distribution," TRW Systems Group, TRW Report No. 2260.19-73D-15(Rev 1), June 1973.

9. J. Tal, G. H. Barber, and L. K. Timothy, "Development and Evaluation of Software Reliability Estimators," Proceedings of the Tenth Hawaii International Conference on System Sciences, January 6-7, 1977, pp. 230-233.

10. E. H. Forman, et al, "An Empirical Stopping Rule for Debugging and Testing Computer Software," George Washington University, NTIS No. AD-A016027, August 18, 1975.

11. E. C. Nelson, "A Statistical Basis for Software Reliability Assessment," TRW-SS-73-03, March 1973.

12. J. R. Brown and M. Lipow, "Testing for Software Reliability," Proceedings 1975 International Conference on Reliable Software, April 21-23, 1975, IEEE Catalog No. 75, CH0940-7CSR, pp. 518-527.

13. F. Akiyama, "An Example of Software System Debugging," IFIP Congress 71, Ljubljana, August 1971, pp. 37-42.

14. M. Lipow and T. A. Thayer, "Prediction of Software Errors," Proceedings 1977 Annual Reliability and Maintainability Symposium, January 1977, pp. 489-494.

15. M. A. Herndon and J. Lane, "An Approach to the Quantification of Software Errors as a Function of Module Complexity," Proceedings of the Tenth Hawaii International Conference on System Sciences, Jan 6-7, 1977, pp. 265-268.

16. R. J. Flynn, "On the Smallest Number of Program Modules Needed to Duplicate Dynamic Independent Interactions," Record 1973 IEEE Symposium on Computer Software Reliability, New York, April 1973, pp. 65-69.

17. J. E. Sullivan, "Measuring the Complexity of Computer Software," MITRE Corp., MTR-2648, June 1973.

18. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

19. M. Lipow, "Estimation of Software Package Residual Errors," TRW-SS-72-09, Redondo Beach, California, November 1972.

20. D. L. Parnas, "The Influence of Software Structure on Reliability," Proc. 1975 Int. Conf. Reliable Software, April 1975, pp. 358-362.

21. P. Wegner, "Abstraction--A Tool in the Management of Complexity," Proceedings 4th Texas Symposium Computation, November 1975.

22. E. W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," Software Engineering, P. Naur and B. Randel, Eds. NATO, Jan. 1969.

23. P. Naur and B. Randal, Ed., Software Engineering Techniques, Report on NATO Conference, October 1968.

24. J. N. Buxton and B. Randal, Ed., Software Engineering Techniques, Report on NATO Conference, October 1969.

25. H. D. Mills, "Software Development," IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, December 1976, pp. 265-273.

26. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles and Goals," Computers, May 1975, pp. 17-27.

27. B. W. Boehm, et al, "Characteristics of Software Quality," TRW-SS-7309, December 28, 1973.

28. B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," 2nd International Conference on Software Engineering, October 13-15, 1976. pp. 592-605.

29. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," CACM, Vol. 15, December 1972, pp. 1053-1058.

30. A. Cohen, "Modular Programs: Defining the Module," Datamation, Vol. 18, January 1972, pp. 34-37.

31. J. B. Dennis, "Modularity," Advanced Course on Software Engineering, Berlin.Heidleburg.New.York, 1973, pp 128-182.

32. B. H. Liskov, "A Design Methodology for Reliable Software Systems," Proceedings 1972 FJCC, pp. 191-199.

33. G. J. Myers, "Characteristics of Composite Design," Datamation, Vol. 19, September 1973, pp 100-102.

34. J. B. Goodenough and D. T. Ross, "System Organization Methodology: An Analysis of Modularity," MAIDS Information Dynamics Technology Requirements Study, NTIS Document No. AD768898/9, June 1973.

35. J. B. Goodenough and R. Zara, "The Effects of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis," NTIS Document No. AD787307/8, July 1974.

36. C. L. McGowan and J. R. Kelly, Top-down Structured Programming Techniques, Petrocelli/Charter, New York, 1975.

37. Programming Conventions and Standards, Developed by SPERRY UNIVAC for the Marine Air Traffic Control and Landing System, Contract: N003228-75-A-4223, August 11, 1976.

38. G. M. Weinberg, The Psychology of Computer Programming, New York: Van Nostrand Reinhold, 1971.

39. M. Woodger, "On Semantic Levels in Programming," Proceedings IFIP Congress 71, August 1971, pp. 79-83.

40. Tom Gilb, Software Metrics, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1977.

41 F. R. Richards, Computer Software: Testing, Reliability Models, and Quality Assurance, NTIS No. AD/A-001260, Navy Postgraduate School, Monterey, California, July 1974.

42. Z. Jelinski and P. B. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation, Edited by Walter Freiberger, Academic Press, 1972, pp. 465-484.

43. M. L. Shooman, "Probabilistic Models for Software Reliability Predictuon," in Statistical Computer Performance Evaluation, Ed. by W. Freiberger, Academic Press, New York, 1972, pp. 485-502.

44. M. L. Shooman, "Probabilistic Models for Software Reliability Prediction," 1972 International Symposium on Fault-Tolerant Computing, Newton Massachusetts, June 19-21, 1972, pp. 211-213.

45. M. L. Shooman, "Software Reliability: Measurement and Models," Proceedings 1975 Annual Reliability and Maintainability Symposium, Washington, D. C., January 28-30, 1975.

46. J. D. Musa, "A Theory of Software Reliability and its Application," IEEE Transactions Software Engineering, September 1975, pp. 312-327.

47. N. Schneidewind, "A Model for the Analysis of Software Reliability and Quality Control," Presented at the 43rd National Meeting of ORSA, May 1973.

48. D. Tsichritzis and A. Ballard, "Software Reliability," INFOR, Vol. 11, No. 2, June 1973, pp. 113-124.

49. R. W. Wolverton and G. J. Schick, "Assessment of Software Reliability," TRW-SS-73-04, September 1972.

50. T. A. Thayer, et al, "Software Reliability Study," Prepared by TRW Defense and Space Systems Group, Redondo Beach, California, for Rome Air Development Center, RADC-TR-76-238, August 1976.

51. E. C. Nelson, "Software Reliability," TRW-SS-75-05, November 1975.

52. J. R. Brown and M. Lipow, "Testing for Software Reliability," TRW-SS-75-02, January 1975.

53. H. T. Nagle Jr., D. B. Kimpsey, G. L. West, and D. B. Bain, "Application of Reliability Techniques to BMD Software," AU-EE-75-C-0034-1, October 15, 1975.

54. A. N. Sukert, "An Investigation of Software Reliability Models," Draft copy from RADC, August 1976.

55. I. A. Miamoto, "Software Reliability in On-line Real Time Environment," Proceedings of the International Conference on Reliable Software, Los Angeles, CA, April 21-23, 1975, pp. 194-203.

56. W. L. Wagoner, "The Final Report on a Software Reliability Measurement Study," Report No. TOR0074(4112)-1, The Aerospace Corp., El Segundo, CA, December 1973.

57. B. Littlewood, "A Reliability Model for Markov Structured Software," Proceedings of 1975 International Conference on Reliable Software, April, 1975, pp. 204-207.

58. B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," Journal of the Royal Statistical Society, Series C, Applied Statistics, 1973, pp. 332-346.

59. G. R. Hudson, "Program Errors as a Birth-and-Death Process," System Development Corporation SP-3011, December 1967.

60. N. Schneidewind, "An Approach to Software Reliability Prediction and Quality Control," Fall Joint Computer Conference, 1972, pp. 837-847.

61. N. Schneidewind, "Analysis of Error Processes in Computer software," Proceedings 1975 Conference on Reliable Software, April 21-23, 1975, IEEE Catalog No. 75, CH040-7CSR, pp. 337-346.

62. M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," Proceedings 1975 Conference on Reliable Software, April 21-23, 1975, IEEE Catalog No. 75, CH040-7CSR, pp. 347-357.

63. T. A. Thayer, "Understanding Software Through Analysis of Expirical Data," Proceedings Nat. Computer Conference, 1975, pp. 335-341.

64. C. R. Litecky and G. B. Davis, "A Study of Errors, Error-Proneness, and Error Diagnosis in COBOL," CACM, Vol. 19, No. 1, January 1976, pp. 33-37.

65. E. A. Youngs, "Error-Proneness in Programming," Doctoral Thesis in Computer Science, University of North Carolina 1970.

66. A. B. Endres, "An Analysis of Errors and Their Causes in System Programs," IEEE Transactions Software Engineering, June 1975, pp. 140-149.

67. D. E. Wright, "An Automated Data Collection System Used for The Study of Software Reliability," Masters Thesis in Computer Engineering, Auburn University, March 1977.

68. W. W. Hines, Probability and Statistics in Engineering and Management Science, The Ronald Press Company, New York, 1972.

69. A. J. Barr, J. H. Goodnight, J. P. Sall, and J. dt. Helwig, A Users Guide to SAS.76, SAS Institute, 1976.

70. I. Bazovsky, Reliability Theory and Practice, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1961.

71. M. L. Shooman, Probabilistic Reliability An Engineering Approach, McGraw-Hill, New York San Francisco Toronto London Sydney, 1968.

APPENDIX A

Hardware Reliability Concepts

A brief summary of the more important concepts associated with the underlying mathematical reliability theory as applied to hardware is presented for those readers unfamiliar with reliability.

If a more detailed insight is desired then the reader should seek other references such as [69-71].

Introduction

The reliability of a component is defined as the probability that the component will function within specified limits for a specified period of time under specified environmental conditions. The frequencies at which components fail per unit time is called failure rate. Its reciprocal value is called mean-time-to-failure, abbreviated as MTTF. Several probability distributions are employed in the study of reliability.

Time-to-Failure Distribution

Let $f(t)$ be the probability density of the time to failure or malfunction of a component; that is, the probability that the component will fail between times t and $(t+\Delta t)$ is given by $f(t).\Delta t$. The probability that the component will fail sometimes before t is given by

$$F(t) = \int_0^t f(t)dt$$

which is sometimes called the "unreliability" function.

Reliability Function

The probability that a component will survive to time t is given by the reliability function

$$R(t) = 1 - F(t).$$

The reader should note the relationships between f(t), F(t) and R(t). In particular

$$f(t) = dF/dt = -dR/dt.$$

Mean-Time-To-Failure

A measure of effectiveness often required in reliability is the MTTF. This is found by taking the first moment of the mean of the time to failure distribution. In terms of the density f(t),

$$\text{MTTF} = \int_0^\infty tf(t)dt.$$

An equivalent expression giving the MTTF in terms of the reliability function is

$$\text{MTTF} = \int_0^\infty R(t)dt.$$

Instantaneous Failure Rate

The probability that a component will fail in the interval from t to t+Δt, given that it has survived to time t, is as follows:

$$P(t < T \leq t + \Delta t | T \geq t) = \frac{F(t + \Delta t) - F(t)}{R(t)}$$

dividing this expression by Δt yields an average rate of failure in the interval from t to t+Δt, given that it has survived to time t, as follows:

$$\left[\frac{F(t + \Delta t) - F(t)}{\Delta t}\right] \cdot \frac{1}{R(t)}$$

By taking the limit of the last expression as $\Delta t \rightarrow 0$, the instantaneous failure rate or hazard function H(t) is obtained; that is,

$$H(t) = \lim_{\Delta t \to o} \left[\frac{F(t+\Delta t-F(t)}{\Delta t}\right] \cdot \frac{1}{R(t)} = \left(\frac{dF(t)}{dt}\right) \cdot \frac{1}{R(t)}$$

using the identities involving f(t), F(t) and R(t) we get the following equivalent expressions for H(t):

$$H(t) = f(t)/R(t)$$

$$= - \frac{dR(t)/dt}{R(t)}$$

$$= - \frac{d}{dt} \ln [R(t)].$$

This differential equation is solved for R(t) to yield

$$R(t) = \exp\left(-\int_0^t H(t)dt\right),$$

and since $H(t) = f(t)/R(t)$ we get

$$f(t) = H(t)\exp\left(-\int_0^t H(t)dt\right) \qquad \text{(A1)}$$

Expression (A1) shows that the time to failure density is related to the instantaneous failure rate function. Also (A1) is a general expression that applies to any type of failure density and hazard rate functions. Figure 21 shows a typical hazard function as a function of age.

The Exponential Model

There are situations where a component reaches a point in its life cycle where the failure rate is constant, that is,

$$H(t) = c, \quad c > 0.$$

On substituting into equation A1 we get the time-to-failure density

$$f(t) = c\exp(-ct), \qquad t > 0$$

which is the exponential probability density function, see Figure 22. Further calculations show that

$$R(t) = \exp(-ct)$$

and

$$MTTF = 1/c.$$

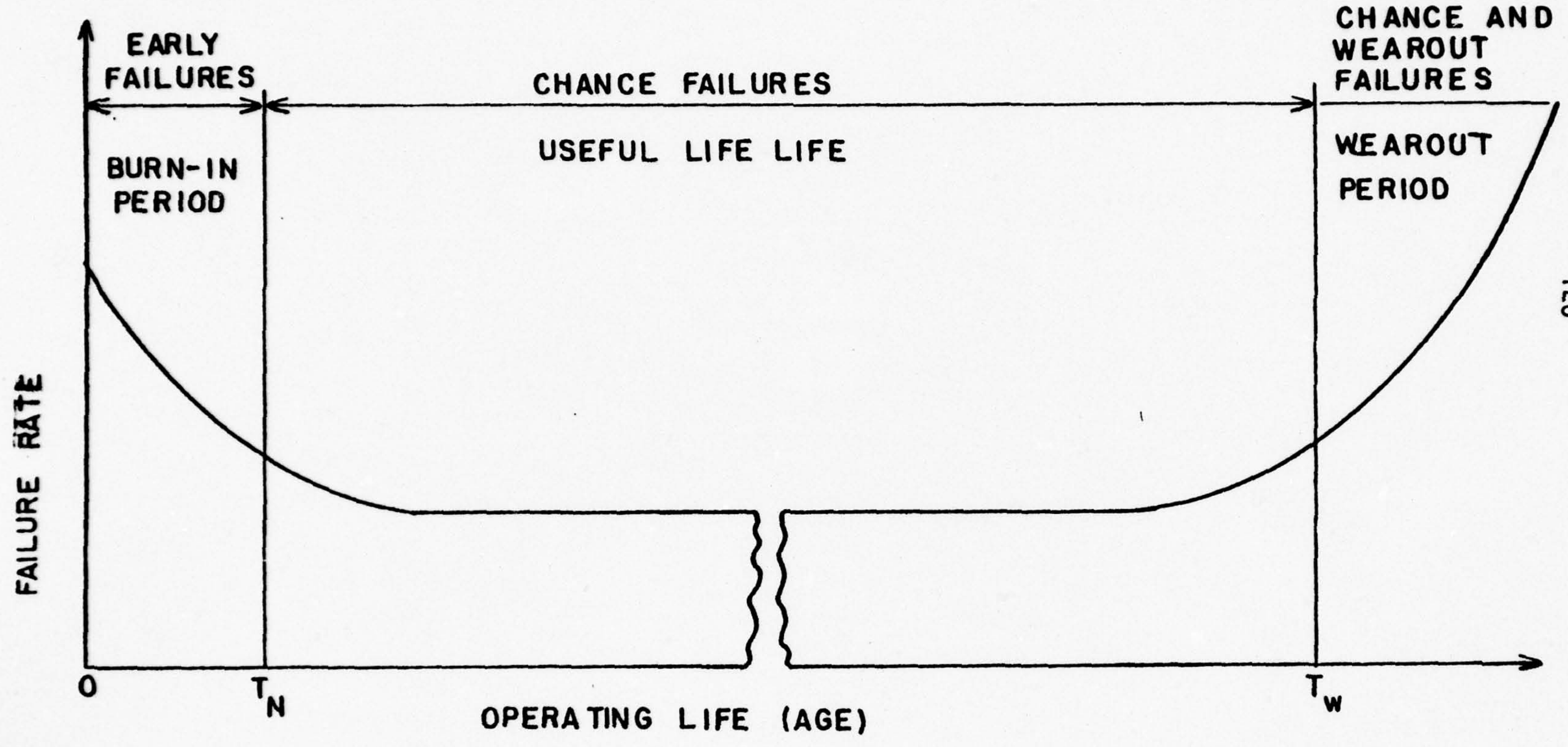
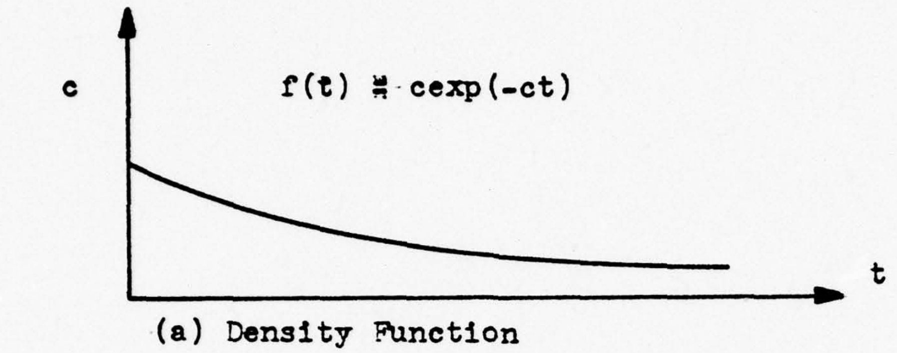
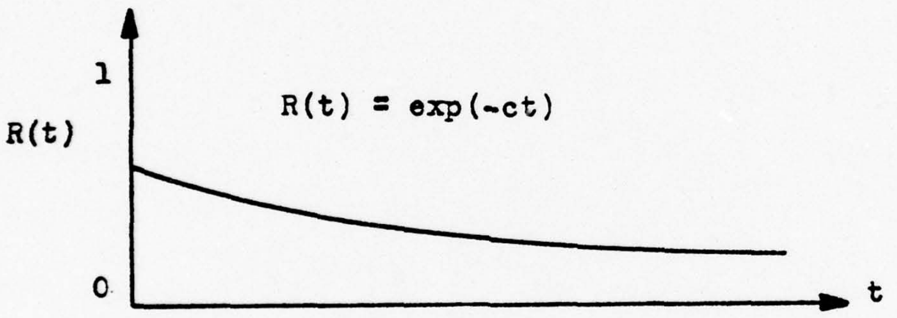


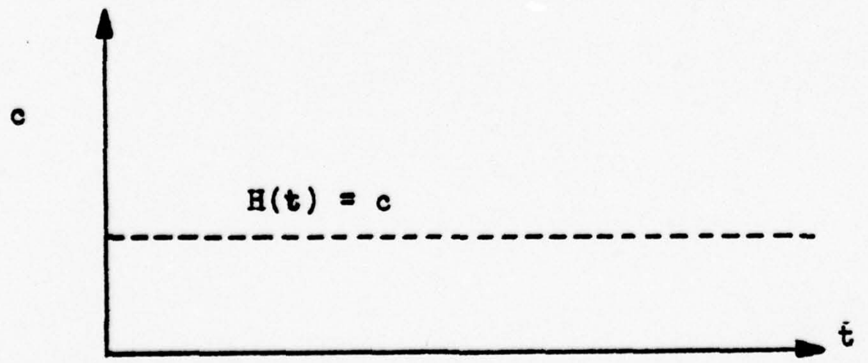
Figure 21. COMPONENT FAILURE RATE AS A FUNCTION OF AGE



(a) Density Function



(b) Reliability Function



(c) Hazard Function

Figure 22. Exponential Distribution

The exponential model is the most widely applied model in reliability engineering. This is due to its simplicity and its theoretical properties such as constant failure rate and loss of memory property [70,71]. In many situations failures are described quite well by the exponential model, but there are also many examples where it is not appropriate.

The Weibull Model

The exponential distribution is a single parameter distribution which can be represented as a special case of a more general two-parameter distribution called the Weibull distribution.

The assumption of a constant failure rate is often appropriate for describing chance failures, but it is not always sufficient. This is particularly true during the early "burn-in" period and the late "wear-out" period in the life cycle of a component, see Figure 21. Nor would the constant failure rate be appropriate during a period of reliability growth due to improvements in the component. Thus, it is obvious that a function that allows an increasing or decreasing failure rate is required. The versatile Weibull function is often used to approximate such failure rates. The hazard function is

$$H(t) = \lambda B t^{B-1} \quad ; \quad t > o \quad ; \quad \lambda B > o \ .$$

When $B < 1$ the failure rate decreases with time; if $B > 1$ it increases with time; and if $B = 1$ the failure rate is constant, see Figure 23. The distribution and density functions are

$$f(t) = \lambda B t^{B-1} \exp(-\lambda t^{B}), \quad t > o$$

$$F(t) = \int_{o}^{t} \lambda B t^{B-1} \exp(-\lambda t^{B}), \quad t > o .$$

The reliability function is

$$R(t) = \exp(-\lambda t^{B}) \quad .$$

The Weibull model enjoys widespread use because it can be justified theoretically and because it is so versatile.

Others Models

There are other probability functions which are useful for describing the random nature of failures. They are the normal, gamma, and lognormal.

$$\text{NORMAL:} \quad f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^{2}} \quad , -\infty < t < +\infty$$

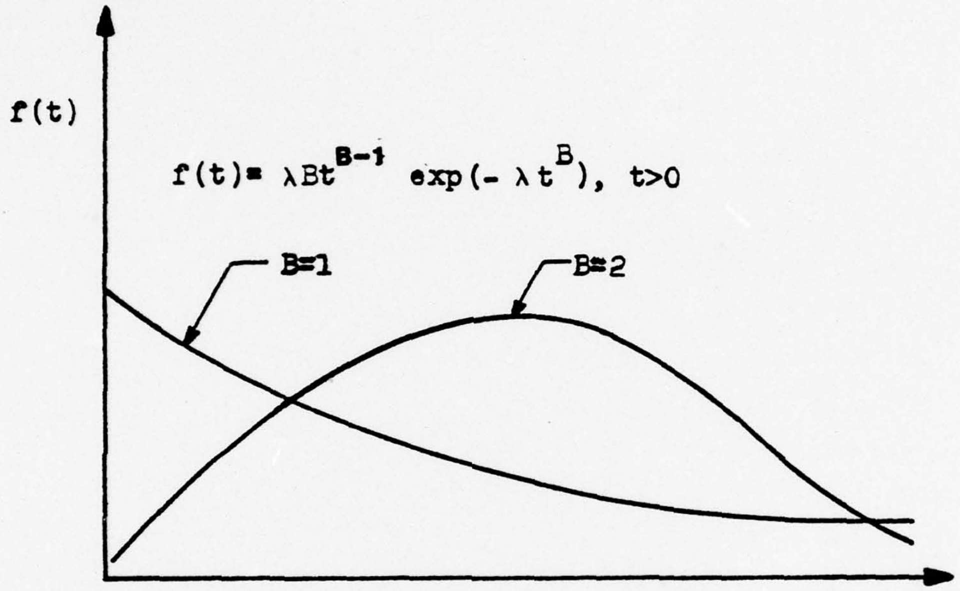


Figure 23. Weibull Density for B=1 and B=2

and

$$\text{GAMMA:} \quad f(t) = \frac{t^{\alpha-1}\exp(-t/B)}{\Gamma(\alpha)B^{\alpha}} \; ; \quad t, \alpha, B > o$$

and

$$\text{LOGNORMAL:} \quad f(t) = \frac{1}{\sqrt{2\pi}\,Bt} \exp[-(\ln t-\alpha)^2/2B^2] \; ; \quad B > o \; .$$

For appropriate choices of parameters, the gamma and lognormal functions can be made to represent increasing or decreasing failure rates. The normal function is primarily used during the wearout period of a component, see Figure 21.

APPENDIX B

Summary of Project 1 Characteristics

Description:

Project 1 is a data collection system. The system provides an on-going data base for input into reliability models. The data base also contains program characteristics as discussed in this paper. The system applies to COBOL programs designed to execute on the Honeywell H6060 computer system throughout the Air Force. The 5 programs in this system utilize a file management system available on the H6060.

Development Agency:

Air Force Data System Design Center; Gunter AFS, Montgomery, Ala.

Computer System: Honeywell H6060.

Operating Mode: Batch.

Number of Programs: 5.

Language: COBOL

Total Number of Lines of Source Code: 2280.

Best Single Variable Characteristics Metric Model:

$\bar{N} = -1.683 + 1.047UBR$.

Best Single Variable Complexity Model

$\bar{N} = 0.287 + 0.382CFC$.

Best Multiple Variable Complexity Model

$$\bar{N} = 3.156 + 0.326CFC - 0.017IOC + - 0.146IC$$

APPENDIX C

Summary of Project 2 Characteristics

Description:

Project 2 is an on-line system involving several kinds of data processing activities such as personnel management, accounting and finance, inventory etc. Only 14 programs are available for analysis.

Development Agency:

City of Montgomery Housing Authority, Montgomery, Ala.

Computer System: National Cash Register NCR8200.

Operating Mode: On-line.

Number of Programs: 14.

Language: COBOL.

Total Number of Lines of Source Code: 19045.

Best Single Variable Characteristics Metric Model:

$$\bar{N} = 3.731 + 0.319LC.$$

Best Single Variable Complexity Model

$$\bar{N} = 8.943 + 0.096CFC.$$

Best Multiple Variable Complexity Model

$$\bar{N} = -1.291 + 0.079CFC + 0.019IOC + 0.314DUC + 0.208COC + 0.005DHC + 0.056IC - 0.074SC$$

APPENDIX D

Summary of Project 3 Characteristics

Description:

Project 3 represents an initial delivery of a large on-line Command Manpower Data System(CMDS). CMDS is a resource accounting and management information system which supports the Manpower and Organization function at Major Command level throughout the Air Force. The programs perform a wide variety of data processing activities, general purpose utility, data retrieval, data maintenance, etc. The programs utilize a file management system available on the H6060.

Development Agency:

Air Force Data System Design Center; Gunter AFS, Montgomery, Alabama.

Computer System: Honeywell H6060.

Operating Mode: On-line and batch.

Number of Programs: 46.

Languages: COBOL, FORTRAN, and Assembler (only COBOL programs analyzed but CALLS to, and interface errors with, FORTRAN and assembly language programs were counted).

Total Number of Lines of Source Code: 54116.

Best Single Variable Characteristics Metric Model:

$\bar{N} = 4.355 + 0.038LC.$

<u>Best Single Variable Complexity Model</u>

$$\bar{N} = 4.069 + 0.024CFC.$$

<u>Best Multiple Variable Complexity Model</u>

$$\bar{N} = 3.094 + 0.022CFC + 0.008IOC + 0.044DUC - 0.017COC$$
$$+ 0.009DHC + 0.005IC - 0.005SC$$

APPENDIX E

Summary of Project 4 Characteristics

Description:

This Project is a combination of Projects 1, 2 and 3.

Development Agencies:

Air Force Data System Design Center; Gunter AFS, Montgomery, Alabama and the City of Montgomery Housing Authority.

Computer Systems: H6060 and NCR8200.

Operating Modes: On-line and Batch.

Number of Programs: 65.

Languages: COBOL, FORTRAN, and Assembly.

Total Number of Lines of Source Code: 75441.

Best Single Variable Characteristics Metric Model:

$\bar{N} = 8.807 + 0.065UBR.$

Best Single Variable Complexity Model

$\bar{N} = 6.789 + 0.254COC$

Best Multiple Variable Complexity Model

$$\bar{N} = 2.845 + 0.029CFC + 0.093IOC + 0.495DUC + 0.119COC - 0.007DHC + 0.028IC - 0.138SC$$

APPENDIX F

Application to Software Reliability

Several models for predicting the number of errors in COBOL programs are presented in this paper. The "best" multiple variable complexity model for Project 2 is used in this appendix. The equation is

$$\bar{N} = -1.291 + 0.079CFC + 0.019IOC + 0.314DUC + 0.208COC + 0.005DHC + 0.056IC - 0.074SC, \qquad (F1)$$

for the range of source data values contained in Table 6. But, how does equation (F1) apply to software reliability? Basically, this equation offers a solution to the primary problem in software reliability, that is predicting the number of errors in a program. Once this number is known, the hazard, time-to-failure distribution, and reliability functions can be derived. Also, the failure frequency and mean-time-to-failure can be calculated, and stopping points for testing programs can be established. To demonstrate how this is done, the Jelinski and Moranda (JM) [42] model (the hazard function was discussed in Chapter III) is used. The basic assumtions of the JM model are:

1) The amount of debugging time between error occurrences has an exponential distribution with an error occurrence

rate or hazard function proportional to the number of errors remaining.

2) Each error discovered is immediately removed, thus decreasing the total number of errors by one.

3) The failure rate between errors is constant.

The hazard function is

$$\begin{aligned} H[t(i)] &= K[N - (i-1)] \\ &= K[N-n] \end{aligned} \qquad \text{(F2)}$$

where

N is the total number of initial errors in a program,

K is the proportionality constant,

t(i) is the i-th time debugging interval, i.e., the time between the i-th and the (i-1)-st errors discovered, and

n is the total number of errors found to date.

As was pointed out in Chapter III, the critical problem is to estimate N and K. Equation (F1) is used to estimate N and the following equation [42] is used to estimate k:

$$\bar{K} = \frac{n}{\bar{N}T - \sum_{i=1}^{n} (i-1)\, t(i)} \qquad \text{(F3)}$$

where

n is the numer of errors found to date

and

$T = \sum_{i=1}^{n} t(i)$ is the total test time from start of testing.

From this one can obtain the time-to-failure distribution (density function)

$$f(t) = \bar{K}\ (\bar{N} - n)\ \exp[-\bar{K}(\bar{N} - n)\ t(i)]. \qquad (F4)$$

The reliability function is

$$R[t(i)] = \exp[-\bar{K}(\bar{N} - n)t(i)]. \qquad (F5)$$

The mean-time-to-failure (MTTF) is

$$MTTF = \frac{1}{\bar{K}\ (\bar{N} - n)}. \qquad (F6)$$

<u>Example Calculations</u>

Program number (observation) 4 from Project 2 is used to illustrate the calculations. The observed number is 22 (see Table 6). It is assumed that 5 errors have been detected during 8 days of testing. Therefore, "n" is 5 and "T" is 8 time units. Each time unit is 1 day. The procedure for calculating reliability equations is:

1) Calculate $\bar{N}$ using equation (F1),
2) Calculate $\bar{K}$ using equation (F2), and
3) Calculate reliability statistics using reliability equations (F4), (F5), and (F6).

<u>Calculating Distributions</u>

1) For $\bar{N}$ where (see Table 6)

CFC = LC + UBR + STOP = 59 + 59 + 1 = 119

IOC = IO = 57

DUC = DR/TD = 6.3474

COC = CO = 53

DHC = DH = 124

IC = OSC + CC + PC = 0 + 0 + 18 = 18

SC = (PAR - EXIT) + 1 = (22 -9) + 1 = 14

$$\bar{N} = -1.291 + 0.079\ (119) + 0.019\ (57) + 0.314\ (6.3474) + 0.208\ (53) + 0.005\ (124) + 0.056\ (18) - 0.074\ (14)$$

$$= 22.8$$

$= 23$ since $\bar{N}$ is an integer.

2) For $\bar{K}$ where

$n = 5$, $T = 8$ and $t(5) = 1$

$$\bar{K} = \frac{n}{\bar{N}T - \sum_{i=1}^{n}(i - 1)\ t(i)} = \frac{5}{(23)(8) - [0 + 1 + 2 + 3 + 4]}$$

$$= \frac{5}{184 - 10} = \frac{5}{174} = 0.0287/\text{day}.$$

3) The hazard function is

$H(t) = \bar{K}\ (\bar{N} - n) = 0.0287\ (23 - n)$.

A failure curve for different values of n is shown in Figure 24.

4) The density function is

$$f[t(i)] = \bar{K}\ (\bar{N} - n)\ \exp[-\bar{K}\ (\bar{N} - n)\ t(i)].$$

$$= 0.0287(23 - n)\exp[-0.0287(23 - n)t(i)].$$

5) The reliability function is

$$R[t(i)] = \exp[-0.0287(23 - n)\ t(i)].$$

Reliability curves for different values of n are plotted in Figure 25. A few calculations were t(i) = 1 and n varies are shown below:

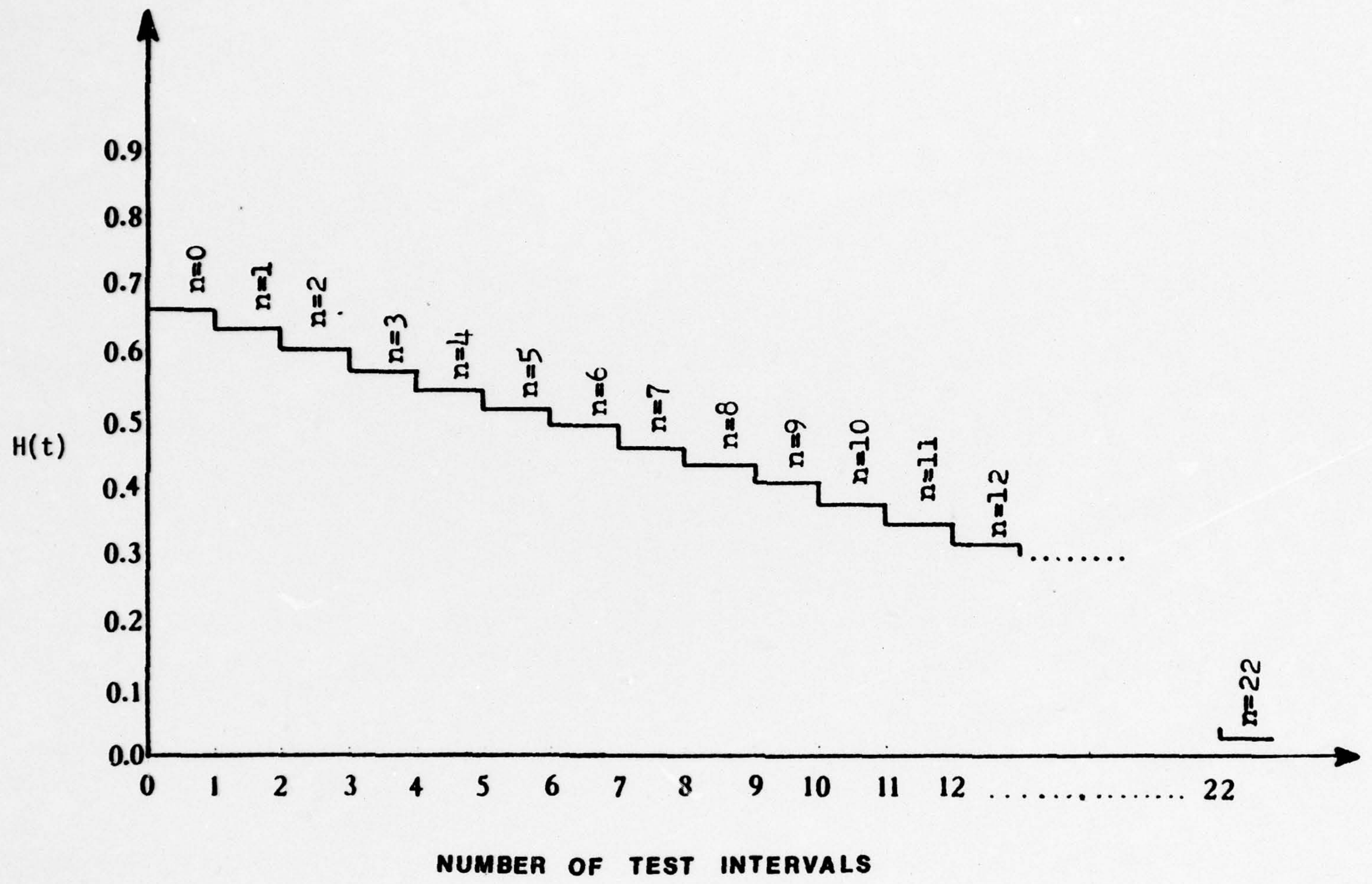


Figure 4. Failure Curve for Program 4 of Project 2

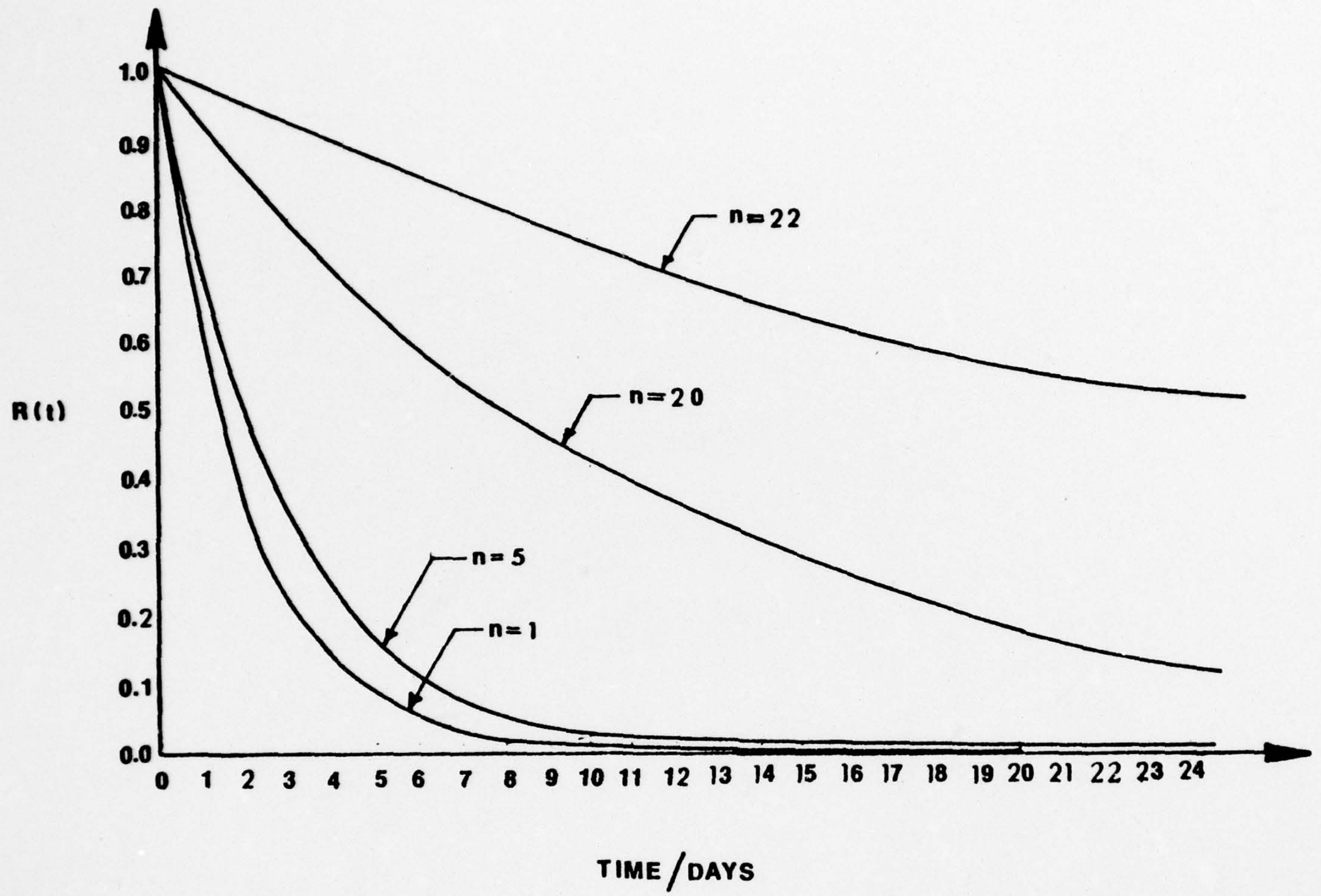


Figure 5. Reliability Curves for Program 4 of Project 2

If t(i) = 1 and n = 5 then

$$R(1) = \exp[-0.0287(23-5)1]$$
$$= \exp[-0.0287(18)1]$$
$$= 0.5966.$$

If t(i) =1 and n= 10 then

$$R(1) = \exp[-0.0287(23-10)1]$$
$$= \exp[-0.0287(13)1]$$
$$= 0.6886.$$

If t(1) = 1 and n = 22 then

$$R(1) = \exp[-0.0287(23-22)\ 1]$$
$$= \exp[-0.0287(1)\ 1]$$
$$= 0.9717.$$

6) The mean-time-to-failure is

$$MTTF = \frac{1}{\bar{K}\ (\bar{N} - n)} = \frac{1}{0.0287(23 - n)}$$

If n = 5 then

$$MTTF = \frac{1}{0.0287(23-5)} = \frac{1}{0.0287(18)} = \frac{1}{0.5166} = 1.94 \text{ days.}$$

Establishing a Stopping Point for Testing

A cut off rule for determining when to stop testing is simply when the reliability of the program reaches a desirable reliability for a specific time period. Let one assume that it is necessary for program 4 to operate 1 day with a reliability of 0.9, When should one stop testing the program? The answer is after n errors have been removed. Calculations for determining this number are shown below:

$R[t(i)] = \exp[-0.0287(23-n)t(i)]$

$0.9 = \exp[-0.0287(23-n)(1)]$

$0.9 = \exp[-0.6601 + 0.0287n]$

$\ln 0.9 = [-0.6601 + 0.0287n]$

$-0.105 = -0.6601 + 0.0287n$

$-0.0287n = -0.6601 + 0.105$

$-0.0287n = -0.5551$

$n = 19.34$

$n = 20$, since n is an integer.

Since $R = 0.9175$ for $n = 20$, one should stop testing the program after 20 errors have been removed.

The next question that one naturally asks is, "approximately how long will it take to remove 20 errors?" Assuming systematic testing procedures are used, a rough estimate is calculated by multiplying the MTTF by the number of remaining errors in the program.

If 5 errors have already been removed then

MTTF(20-5) = 1.94(15) = 29.1 days.

This estimate should be updated as errors are removed from the program.